

数据中心网络中的应用速率和延 迟优化

(申请清华大学工学博士学位论文)

培养单位:计算机科学与技术系
学 科:计算机科学与技术
研 究 生:张 晗
指 导 教 师:尹 霞 教 授

二〇一八年三月

Throughput and latency optimization for applications in data center networks

Thesis Submitted to
Tsinghua University
in partial fulfillment of the requirement
for the professional degree of
Doctor of Engineering

by
Han Zhang
(Computer Science and Technology)

Thesis Supervisor : Professor Xia Yin

March, 2018

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：(1) 已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；(2) 为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

(保密的论文在解密后应遵守此规定)

作者签名：_____

导师签名：_____

日 期：_____

日 期：_____

摘要

在过去的几年中，越来越多的企业有了自己的数据中心，还出现了比如 Amazon，微软和谷歌这类数据中心服务提供商。在数据中心如何使用廉价，常见的网络设备来给数据中心应用提供低延迟，高带宽服务是十分重要的问题。TCP 是 Internet 中应用最多的传输协议。尽管 TCP 被广泛的应用在数据中心，但是在数据中心中使用 TCP 存在以下问题：（1）TCP 使用基于丢包的拥塞控制方法，当网络中出现拥塞时，发送端发送速率震荡大，网络利用率低。（2）交换机的队列长，排队延迟高。（3）应用的数据流均分网络带宽，不能满足不同应用对带宽和延迟不同的需求。（4）TCP 是数据流级别的传输，难以满足数据中心任务级别传输要求。

因此，需要对数据中心应用传输同时做流级别和任务级别的传输优化。对流级别和任务级别的传输，应同时兼顾网络拥塞和应用语义特征。基于此，本文从流和任务级别对数据中心传输做出优化，主要研究内容和贡献如下：

（1）提出了数据中心传输模型，具体包括了基于 ECN 的流传输模型和任务级别传输优化模型。通过基于 ECN 的流传输模型，可以推算出基于 ECN 流传输协议的参数设置区间。针对数据中心任务传输，本文提出理想化权重流组完成时间优化 (Idealized Weighted Coflow Completion Time Minimization, 简称 IWCCTM) 问题，针对 IWCCTM 问题，本文提出 2-近似离线算法解决此问题。

（2）针对流级别的优化，本文提出了期限自适应的流传输策略 – LPD 和基于流传输时间的速率控制机制 – FDRC。LPD 和 FDRC 都遵循“越拥塞，越区分”的设计原则。因此即使在重度网络负载情形下，LPD 和 FDRC 依然可以根据应用的紧急程度和网络拥塞对传输进行优化。同时，错失期限的流数目和流完成时间也得到充分优化。

（3）针对任务级别的优化，本文提出基于流信息的调度策略 D-Target 和基于任务重要性和网络拥塞调度策略 Yosemite。其中 D-Target 在可以在预先得知流信息的前提下，同时结合纠删码存储系统的源选择问题，优化文件平均存取时间。Yosemite 可以不预先知道流信息的前提下，优化了任务传输时间。

（4）设计并且实现了传输优化系统 FlyTransfer，并在 OpenStack 和小型数据中心下部署和测试 FlyTransfer。实验发现，FlyTransfer 可以同时进行流级别和任务级别的优化。本文最后对 FlyTransfer 进行性能评估。

关键词：数据中心；网络；应用；延迟；带宽

Abstract

In the past few years, more and more companies have their own data centers. Some companies such as Amazon, Microsoft, and Google even provide data center service. In this case, how to use cheap, common network devices to provide low-latency, high-bandwidth services to the applications of data center is an important problem. TCP is the most widely used transport protocol in the Internet. Although TCP is widely used in the data center, the traditional TCP transport protocol has following problems: (1) TCP's congestion window will be halved when there is congestion in the network and this will lead the network utility at low level. (2) Queue delay of TCP is too long to satisfy the latency demands of applications. (3) TCP is a fair allocation method and fair allocation can not meet the demands of applications on bandwidth and latency. (4) TCP can not meet the requirements of task level transmission.

As a result, it is necessary to optimize the transport of applications from flow level and task level. Also, the different bandwidth and latency demands of applications should also be taken into consideration. Based on these, we make the following contributions in this paper:

(1) We propose flow transfer model and task optimization model. With the ECN-based flow level model, parameters of rate control methods can be analyzed. Based on the task-level model, we propose Idealized Weighted Coflow Completion Time Minimization(IWCCTM) problem. To solve IWCCTM problem, we propose a 2-approximate optimal method to solve.

(2) For the optimization of flow level, we propose the load adaption protocol-LPD and the flow duration time rate control protocol-FDRC. Both LPD and FDRC follow the principle "more load, more differentiation". As a result, even under heavy load, the percentage of flows missing deadline as well as flow completion time can also be optimized.

(3) For the optimization of task level, we propose the centralized schedule methods D-Target and Yosemite. With the information of flows, D-Target can reduce the average task completion time. In this paper, we use the erasure coding storage system to prove this. Yosemite makes an advance, it can optimize the transfer of tasks without the knowledge of flows.

(4) We design and evaluate FlyTransfer, a system that can schedule the transfer of flows as well as tasks. We deploy FlyTransfer at the platform of openstack and a small data center. Evaluations show that FlyTransfer can optimize the transfer of tasks and flows with small overheads. We also test the overhead of FlyTransfer in this paper.

Key words: Date Center;network;application;latency;throughput

目 录

第 1 章 引言	1
1.1 研究背景和研究意义	1
1.2 论文研究内容	2
1.3 论文的主要贡献	5
1.4 论文的组织结构	6
第 2 章 相关文献综述	7
2.1 本章引言	7
2.2 应用传输概述	7
2.2.1 分布式应用的通信模型	7
2.2.2 网络流量特征分析	10
2.2.3 TCP 传输存在的问题	12
2.3 数据中心网络应用传输方案	14
2.3.1 流级别传输优化方案	14
2.3.2 任务级别传输优化方案	17
2.3.3 数据中心传输方案存在的问题	19
2.4 本章小结	20
第 3 章 基于 ECN 标记的流传输模型	21
3.1 本章引言	21
3.2 场景描述	21
3.3 ECN 标记流模型	22
3.3.1 模型描述	23
3.3.2 模型实例	23
3.4 本章小结	25
第 4 章 基于负载自适应原则的传输优化方案	26
4.1 概述	26
4.2 相关工作和研究动机	28
4.2.1 相关工作	28
4.2.2 研究动机	28
4.3 “越拥塞，越区分”的设计原则	30
4.3.1 分析伽马校正函数	30

4.3.2 基于截止期限的速率控制策略设计原则.....	31
4.4 正比负载差分策略 (LPD) 以及分析	32
4.4.1 正比负载差分策略 (LPD).....	32
4.4.2 LPD 分析	33
4.4.3 LPD 拓展	39
4.5 实验验证	39
4.5.1 解决研究动机提出的问题	39
4.5.2 参数选择.....	42
4.5.3 仿真测试.....	42
4.5.4 真实环境下测试	48
4.6 对“越拥塞，越区分”原则的讨论.....	51
4.6.1 不同的惩罚函数	52
4.6.2 使用剩余时间	52
4.7 本章总结	53
第 5 章 基于流持续时间的传输优化方案	54
5.1 概述	54
5.2 相关工作和研究动机	56
5.2.1 相关工作.....	56
5.2.2 研究动机.....	56
5.3 FDRC 策略.....	58
5.3.1 基于流持续时间的速率控制策略	58
5.3.2 FDRC 协议	59
5.3.3 FDRC 分析	59
5.4 实验验证.....	63
5.4.1 解决研究动机提出的问题	63
5.4.2 真实流量和拓扑下仿真	64
5.4.3 真实环境下的测试床测试	72
5.5 本章总结	74
第 6 章 数据中心任务级传输优化模型	75
6.1 本章概述.....	75
6.2 场景描述.....	75
6.2.1 网络模型.....	76
6.2.2 传输任务的定义	76

6.3 传输任务的离线调度模型	76
6.3.1 开放并行商店问题	77
6.3.2 NP-hard 证明	77
6.3.3 离线算法	78
6.3.4 离线算法 $(2 - \frac{2}{n+1})$ -近似证明	78
6.4 本章小结	81
第 7 章 基于流信息的任务级传输优化方案	82
7.1 概述	82
7.2 研究动机和相关工作	84
7.2.1 研究动机	84
7.2.2 相关工作	85
7.3 网络模型和分析	87
7.3.1 问题分析	87
7.3.2 NP-hard 证明	88
7.4 算法和分析	88
7.4.1 SIFATM 离线算法	88
7.4.2 从离线到在线	89
7.4.3 源选择问题	91
7.4.4 D-Target 的设计	92
7.5 实验验证	93
7.5.1 仿真测试	94
7.5.2 真实流量仿真	94
7.5.3 不同设置下性能	95
7.5.4 离线算法和在线算法之间的差异	97
7.6 本章总结	98
第 8 章 基于重要性和网络拥塞的任务传输调度方案	99
8.1 概述	99
8.2 研究动机和相关工作	101
8.3 Yosemite 算法设计	104
8.3.1 解决 IWCCM 的另一个简单离线算法	105
8.3.2 Yosemite 算法	105
8.4 实验验证	107
8.4.1 仿真方法介绍	107

8.4.2 Facebook 流量下仿真测试	108
8.4.3 使用中等规模的数据中心流量数据进行测试	110
8.4.4 权重的讨论	111
8.4.5 策略和最优策略的差距	112
8.4.6 在线策略和离线策略的差距	112
8.5 本章小结	113
第 9 章 数据中心传输系统和性能评估	114
9.1 概述	114
9.2 FlyTransfer 系统	114
9.2.1 FlyTransfer 架构	114
9.2.2 FlyTransfer 组件	115
9.2.3 backup 组件	118
9.3 性能评估	118
9.3.1 任务级优化对比	118
9.3.2 流级优化对比	119
9.3.3 系统开销	120
9.4 本章小结	120
第 10 章 总结和展望	121
10.1 论文总结	121
10.2 进一步研究工作	123
参考文献	125
致 谢	129
声 明	130
个人简历、在学期间发表的学术论文与研究成果	131

第1章 引言

1.1 研究背景和研究意义

云计算技术是 IT 产业界的一场技术革命, 已经成为了 IT 行业发展的一个大方向。各国政府纷纷将云计算服务视为国家软件产业发展的新机遇。美国政府在 IT 政策和战略中也加入了云计算因素, 美国国防信息系统部门 (DISA) 正在其数据中心内部搭建云环境^[1]。作为云计算的载体 – 数据中心, 近些年也收到了越来越多的关注。事实上, 从 2010 年起, 全球数据中心的市场规模一年比一年庞大, 已经从 2010 年的 20 亿美元提高到了 2015 年的 4.6 亿美元, 平均每年增长幅度达到了 14.3%^[2]。所谓数据中心, 就是一套由计算机及相关配套设备所组成的, 以储存、传递、展示、加工处理数据为主要目的的完善系统工程^[2]。现在很多应用都部署在数据中心, 为用户提供各种各样的服务, 例如, 为用户提供计算服务的 Hadoop^①、Spark^②、Flume^③、Storm^④等, 为用户提供存储的 ceph^⑤等。甚至, 很多企业比如国外的 facebook, youtube, 国内的中国银行, 中国电信等, 都有自己的数据中心。数据中心已经成为国内外工业界和学术界关注的重心和研究热点。

在实际中, 数据中心有成千上万台机器。根据统计, 截止 2016 年, 谷歌的数据中心中有多达 45 万台机器^[3]。在数据中心的服务器中, 部署着各种各样的应用, 因为单个机器的计算能力, 容量等有限, 而具有超高计算能力和超高存储能力的大型计算机的成本又太高, 因此, 应用往往采取分布式的方法部署在集群中。部署在集群中的应用数据流族, 有共同的目标和共同的语义, 它们或是计算同一个目标, 或者是作为存储同一个巨型文件, 或者是为了达到同一个优化目标等。各种各样的分布式应用部署在数据中心, 突出了数据中心作为信息服务载体的核心地位, 也同时给数据中心的建设提出了更加严峻的挑战。

数据中心网络 (Data Center Networks, 简称 DCNs) 是指数据中心内部通过高速链路和交换机, 路由器连接大量服务器的网络。传统数据中心网络主要采用层次结构实现, 且承载的主要是客户机/服务器模式的应用。多种应用同时在同一个数据中心内运行, 每种应用一般运行在其特定的服务器/虚拟服务器集合上^[3]。因为数据中心的应用常常分布式的部署在集群中, 而这些应用常常需要多个计算或者

① Hadoop. <http://hadoop.apache.org/>.

② Spark. <http://spark.apache.org/>.

③ Flume. <http://flume.apache.org/>.

④ Storm. <http://storm-project.net/>.

⑤ Ceph. <http://ceph.com>.

存储步骤才能得到预期的结果。数据中心应用内部，应用之间往往进行频繁的通信，数据中心网络，就是这些应用通信的桥梁。作为数据中心中应用传输的媒介，随着数据中心规模的变大，应用对数据中心网络提出了越来越高的要求，网络性能已经逐渐成为数据中心发展的瓶颈。在网络给数据中心提供的服务中，应用获得的带宽和传输的延迟，是评价服务质量高低重要的因素。应用的带宽和传输延迟常常对用户体验产生重要的影响。根据 facebook 的报告^[4]，应用的延迟每增加 100 毫秒，收入会减少 1%。

因此，越来越多的研究，集中在优化数据中心应用的带宽和延迟上。互联网采用 TCP 作为传输协议。直接在数据中心采用 TCP 会导致一系列问题：首先，TCP 采取滑动窗口机制进行控制，当发生拥塞时，交换机会出现丢包，此时发送端的拥塞窗口会减半，从而发送端速率会减小。滑动窗口减半会引发链路震荡大，从而链路利用率低。没有拥塞时，发送端的拥塞窗口会不断增大，发送端速率不断增大，从而导致交换机缓冲区不断变大，引起较大的排队延迟。此外，TCP 是公平分配带宽的策略，然而对于数据中心的应用而言，不同的应用对延迟和带宽的需求是不同的。应用如时钟同步，Memcached，Naiad 等需要数据中心网络提供低延迟，而 hadoop 等计算应用需要数据中心网络提供高带宽。事实上，数据中心的资源总量是一定的，采用 TCP 进行传输，不能有效的满足应用对延迟和带宽的需求，造成数据中心网络资源不能合理高效的使用。当前，越来越多的应用部署在规模日益增大的数据中心集群中，给数据中心网络带来了各种各样的问题和挑战，如何合理的分配数据中心网络资源，使得数据中心能够满足日益增加的网络需求，是当前业界面临的一个重大挑战。

针对上述的问题，本文对数据中心的应用传输做优化，从而应用可以提供更好的用户体验，并可以充分合理的利用信道，提高资源利用率。本文的工作重点是：一是从流级别进行传输优化，改进数据传输协议，使应用可以根据网络拥塞状况和应用的期限等因子调整带宽。从而使得不同需求的应用获得不同大小的带宽。二是为数据中心任务分配优先级，采用集中式的方法，使的网络资源的分配可以根据应用的优先级和网络拥塞进行动态调配。从而可以提高用户体验，同时可以使的网络资源利用率高。需要说明的是，本文提供的方法，不止用于数据中心网络，也可以用于其他类似网络环境下。

1.2 论文研究内容

作者的博士论文主要包括以下几个方面：

(1) 综述了国内外研究现状

本文介绍数据中心应用传输方案的相关工作。首先，从常见应用的通信模型和数据流量等方面对数据中心传输进行概述。本文重点介绍4类通信模型：映射规约模型（Map-Reduce），数据流模型（Dataflow），大型并行数据传输模型（Bulk Synchronous Parallel），分区-聚合模型（partition-aggregate），并介绍了查询流，背景流等流量特征和数据中心应用期限等特征。因为传统的TCP协议，难以满足不同的应用对带宽和延迟的需求，本文从流级别和任务级别对数据中心传输方案进行综述。最后，本文总结数据中心传输方案存在的问题。

(2) 研究了基于 ECN 标记的流传输模型

本文介绍基于 ECN 标记的流模型。首先，对基于 ECN 标记的流模型的场景进行介绍。然后，介绍了基于 ECN 标记的流模型，最后，根据基于 ECN 标记的流模型实例化 DCTCP，得到 DCTCP 流模型，作为 ECN 标记的流模型的一个实例，并使用 ns-2 和流模型进行比对。

(3) 研究了基于负载自适应原则的传输优化方案

现在很多在线应用部署在数据中心，数据中心网络的性能和底层技术吸引了越来越多的研究者的兴趣。为了达到更好的网络性能，近期的研究从不同角度优化数据中心网络流量传输和调度，并且设计各种路由和传输方案。特别是对于必须及时为用户服务的应用，数据流的传输应尽快的完成，有的必须在截止时间之前完成，最近业界出现了一系列的考虑流传输期限的网络流量控制策略。本文主张在设计数据流传输方案时，应该遵循一个简单的原则：拥有不同截止期限的流在其带宽分配和占用上应该被区分，网络负载越重，数据流越被区分。本文认为设计流量控制策略时应该遵循这个原则。根据这个原则，本文并提出了一种简单的拥塞控制算法-正比负载差分（Load Proportional Differentiation，简称 LPD）作为

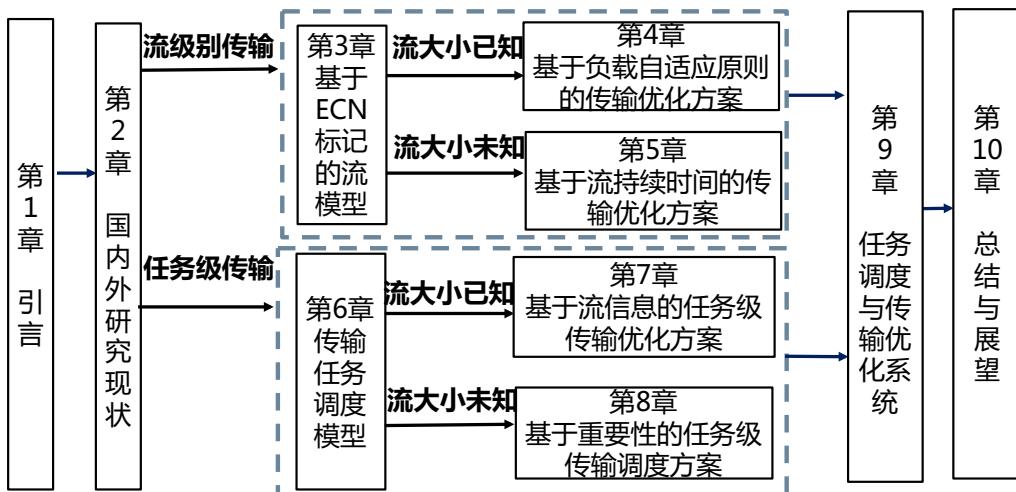


图 1.1 论文结构

其应用。在不同的仿真环境和测试床上，针对不同的网络拓扑和负载情况下评估了 LPD 的性能。

(4) 研究了基于流持续时间的传输优化方案

数据中心现在正成为许多应用（如网络搜索和零售）的部署平台。由于 TCP 不能满足应用程序对延迟和吞吐量的要求，因此提出了许多基于 TCP 的协议（例如，DCTCP, D²TCP, L²DCT）来对 TCP 进行补充。其中，D²TCP 等协议将明确的截止时间纳入拥塞窗口调整过程，以保证流在截止时间之前完成。L²DCT 等协议在计算拥塞窗口调整因子时考虑流量大小，保证短流量的吞吐量和延迟。这两种方法在一定的场景下运行良好，但在两个方面存在一些不足。首先，这两种方法只能减小流错失期限的百分比或者能够减小短流的延迟，但是不能够同时优化这两个目标。其次，这些方法中的大多数都需要用户知道流的信息（例如截止时间，流量大小），这些信息对于一些应用来说，可能很难事先确切得知。本文主张引入流持续时间进入拥塞窗口调整的过程中。基于此，本文提出基于流持续时间速率控制（Flow Duration Rate Control，简称 FDRC）机制。本文发现，在不得知流具体信息的情况下，FDRC 可以达到同时减少流错失期限的比例和减小流平均完成时间。本文从理论上分析了 FDRC 的行为，并在 ns-2 和 Linux 内核实现 FDRC。

(5) 研究了数据中心任务级传输优化模型

本文对数据中心任务传输模型进行介绍，首先，本章从数据中心非阻塞模型，以及传输任务等对数据中心任务级传输的模型进行介绍。随后介绍数据中心任务调度问题的复杂度，最后，引入任务的离线调度算法，并证明离线调度算法的近似度。

(6) 研究了基于流信息的任务级传输优化方案

随着大数据应用的部署，越来越多的数据存储在大数据存储系统中。大数据存储系统希望给用户提供安全，稳定，高效的文件存取服务。纠删码存储系统已被诸如 Google 和 Facebook 等公司广泛使用，因为它使用纠错码来保证文件系统的可靠性。在纠删码存储系统中，使用 (n, k) MDS 纠删码用于将文件分成 n 个块。当用户想要访问文件时，文件系统可以选取 n 个块中的任何 k 个子集来重建文件。在这种情况下，如何从 n 个数据块中选出 k 个数据集，如何高效的传输这 k 个数据块，成为一个重要的问题。

(7) 基于重要性和网络拥塞的任务传输调度方案

传统的网络资源管理机制主要是流级别或者包级别的。最近，流组（coflow）作为一种新的并行应用数据传输通信的抽象模型而提出。流组（coflow）对网络资源的应用级语义进行了有效的建模，因此可以通过将流组（coflow）作为网络资源

分配或调度的基本元素来更好地实现一些高级优化目标，如减少应用程序的传输延迟等。虽然有效的流组（coflow）调度方法已经被研究，但是应用的重要性等因素并没有被考虑，如何根据应用的重要性和 coflow 的特性对应用进行调度是一个重要问题。

(8) 设计并实现和评估了数据中心应用传输优化系统

本文设计并实现了数据中心传输系统 FlyTransfer，并介绍了 FlyTransfer 系统架构和系统的各个组件。随后使用数据中心真实流量，从任务级别调度，流级别优化和系统开销等方面对系统进行性能评估。

1.3 论文的主要贡献

(1) 提出了基于负载自适应原则的传输优化方案

本文主张在设计数据流传输方案时，应该遵循一个简单的原则：拥有不同截止期限的流在其带宽分配和占用上应该被区分开，网络负载越重，数据流越被区分。本文认为设计流量控制策略时应该遵循这个原则。根据这个原则，本文提出了一种简单的拥塞控制算法-正比负载差分（Load Proportional Differentiation，简称 LPD）算法作为其应用。本文在不同的拓扑和负载场景下评估 LPD，并对 LPD 进行仿真和测试床评估。与 D²TCP（最先进的基于期限的拥塞控制策略）相比，使用 LPD，错过最后期限的流比例减少 25% 以上。与 Karuna 相比，LPD 平均有 5% 的性能损失。但是在拥塞严重的情况下，LPD 的性能比 Karuna 高 5% - 10%。事实上“越拥塞，越区分”是一个通用的原则，它也可以用于其他目标的优化，比如优化流平均完成时间。与基于窗口的协议 L²DCT 相比，LPD 可以将流平均完成时间减少 30%。和最先进的流平均完成时间优化策略 pFabric 相比，LPD 的性能比之差 20%。

(2) 提出了基于流持续时间的传输优化方案

在本文中，我们主张使引入流持续时间进入拥塞窗口调整的过程。基于此，我们提出基于流持续时间速率控制机制（Flow Duration Rate Control，简称 FDRC）。本文发现，无需得知流信息的前提下，FDRC 可以同时减少流错失期限的比例并且减小流平均完成时间。本文从理论上分析了 FDRC 的性能，并在 ns-2 和 Linux 内核实现 FDRC。实验表明，在几乎所有场景下，FDRC 比 D²TCP 和 L²DCT 的性能都要好。平均来说，FDRC 比 D²TCP 性能高 30%，比 L²DCT，性能提高大约 10%。

(3) 提出了基于流信息的任务级传输优化方案

本文使用纠删码存储系统为例，进行最小化文件平均访问时间（File Access Time，简称 FAT）的优化。为了实现这一点，本文提出了最小负载优先的启发式算

法做为纠删码存储系统数据源选取的策略。并且从任务级别对传输系统进行优化。在此基础上，本文设计并实现了 D-Target，一个集中调度器，对纠删码存储系统的传输进行调度。最后，通过 AT&T 的存储系统数据验证 D-Target 的性能。结果表明，D-Target 在 FAT 性能方面分别比 TCP，Aalo，Barrat 和 pFabric 分别提高了 2.5×，1.7×，1.8×。

(4) 提出了基于重要性和网络拥塞的任务传输调度方案

本文提出加权流组完成时间 (Weighted Coflow Completion Time，简称 WCCT) 最小化问题，同时针对此问题，设计了不需要知道流大小就可以进行调度的在线算法-Yosemite。Yosemite 可以根据流组权重和网络拥塞程度进行流组调度。然后，本文通过数据中心的真实流量对系统进行仿真评测。实验结果结果显示，与最新的流组调度算法相比，Yosemite 可以使得平均 WCCT 减少超过 40%，对于重要性在平均程度以上的流组，流组的平均完成时间减少超过 30%。和最有效的流组调度方法相比，针对 WCCT 优化的性能大约提高了 30%，对于重要性在平均以上的流组，平均完成时间减少大约 25% ~ 30%。

1.4 论文的组织结构

如图1.1所示，本文总共分为 10 章，其中第 2 章到第 9 章是论文的主体部分。第 1 章，是论文的引言部分。主体部分主要包括流级别的优化，任务级别的优化和应用传输系统的介绍。其中，论文的第 3 章到第 5 章侧重流级别的传输优化。第 3 章介绍的是基于 ECN 标记的流模型，第 4 章介绍的是基于负载自适应原则的传输优化方案，第 5 章介绍的是基于流持续时间的传输优化方案。本文的第 6 到 8 章介绍的是任务级传输优化方法，其中第 6 章介绍的是任务级别速率和延迟优化模型，第 7 章介绍的是任务级的纠删码传输优化系统，第 8 章介绍基于重要性和网络拥塞的任务传输优化。第 9 章是论文的第三部分，主要介绍应用传输系统 FlyTransfer 以及其性能评测。第 10 章是论文的总结部分，主要是对当前工作的总结和对未来工作的展望。

第2章 相关文献综述

2.1 本章引言

本章介绍数据中心应用传输方案的相关工作。首先，本章从常见应用的通信模型和数据流量等方面对数据中心传输进行概述。然后，本章从流级别和任务级别对数据中心传输方案进行综述。最后，本章总结数据中心传输方案存在的问题。

2.2 应用传输概述

在过去的几年中，数据中心随着在线应用的迅速发展取得了巨大的发展。越来越多的企业有了自己的数据中心，还出现了比如 Amazon，微软和谷歌的这些数据中心服务提供商。在数据中心中有一个持续火热的话题：如何使用廉价，常见的网络设备来给数据中心应用提供提供低延迟，高带宽服务。尽管数据中心中存在网络搜索，购物，广告等这些在线应用，这些应用也扮演者不同的角色，但是，这些应用在使用的通信模型，传输延迟等方面有一些共同的特征。

2.2.1 分布式应用的通信模型

大多数集群应用程序的计算框架都是输入特定的配置参数，然后使用特定的计算模型，得到计算结果。计算过程中，常常产生大量的数据，节点之间会进行频繁通信，不同的计算模型常有不同的通信模式，这些通信模式可以总结成不同的通信模型。图2.1显示了6种通信模型，这些通信模型可以分成4类：映射规约模型（Map-Reduce），数据流模型（Dataflow），大型并行数据传输模型（Bulk Synchronous Parallel），分区－聚合模型（Partition-Aggregate）。

2.2.1.1 映射规约模型（Map-Reduce）

MapReduce^[5]是一个众所周知的集群计算框架。如图2.1(a)所示，每个 mapper 都从分布式文件系统(DFS)中读取输入，执行用户定义的计算，并将中间数据写到磁盘上；每个 reducer 都从不同的 mapper 中提取中间数据，将它们合并，并将其输出写入 DFS。

给定 m 个 mappers 和 r 个 reducer，MapReduce 的 shuffle 阶段将产生总数为 $m \times r$ 条数据流。经过 reducer 处理完毕以后，至少有 r 条数据流被复制。MapReduce 模型中 shuffle 阶段会产生很多并行的数据流，直到最后一条数据流传输结束此传

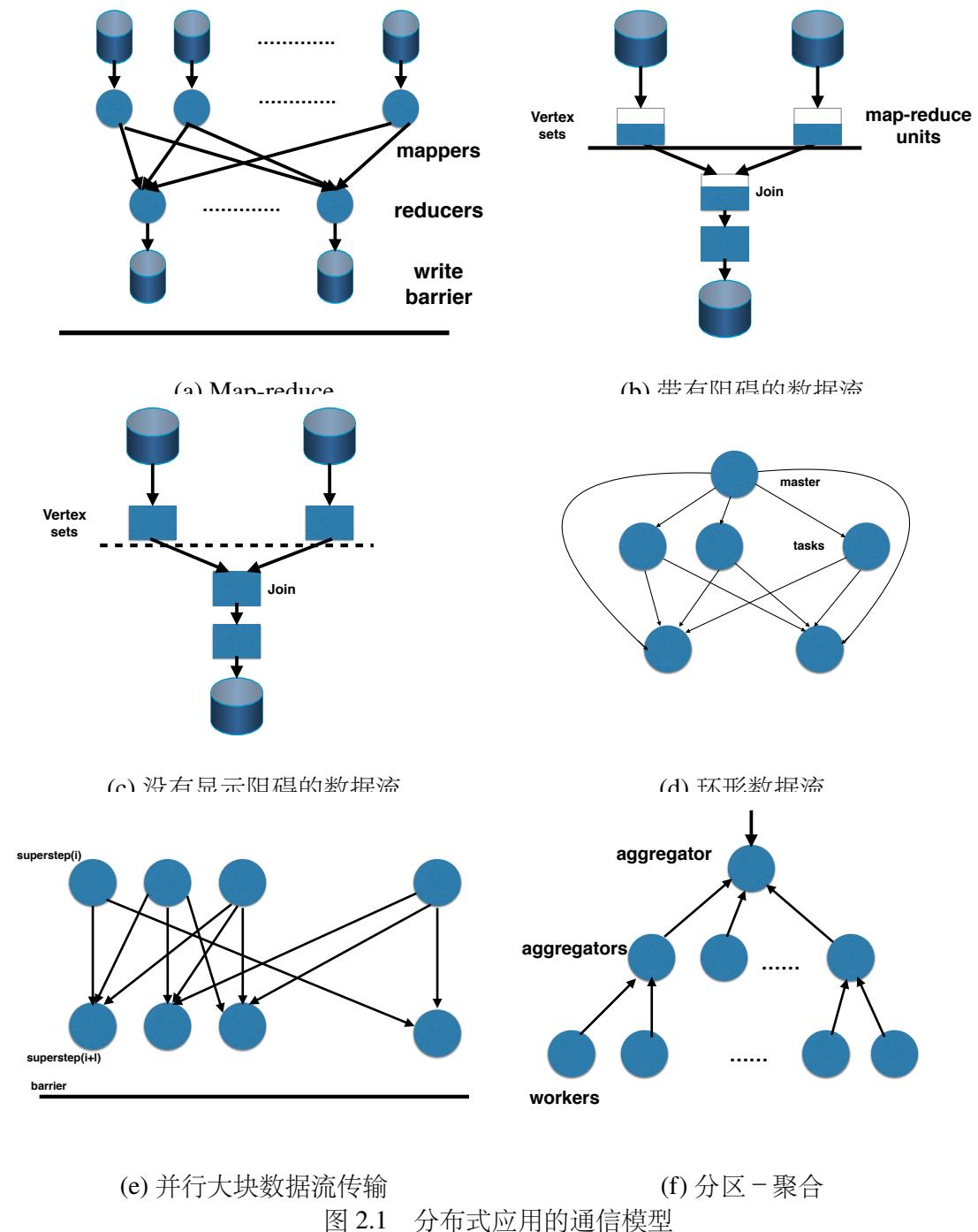


图 2.1 分布式应用的通信模型

输过程才会传输完毕。因此，在任务传输的最后阶段，有一个显式的同步。有研究^[6]针对 MapReduce 存在任务数据流同步的特点，优化了 shuffle 阶段的传输，以此来提高传输的性能。

2.2.1.2 数据流模型 (Dataflow)

虽然 MapReduce 被广泛使用，但是因为其存在数据阻塞等问题，常常会影响应用的整体性能。Dataflow 通信模型的出现解决了 MapReduce 很多的不足，数据流模型有以下三种方式：

带有阻塞的数据流模型。如图2.1(b) 所示，有的多阶段的数据流模型使用 MapReduce 作为其构建模块（例如，Hive^①, Pig^[7]）。因为基于 MapReduce，因此，MapReduce 类似，带有阻塞的 Dataflow 的通信模型在每个构建模块的末尾都有同步操作。

无阻塞的数据流模型。如图2.1 (c) 所示，一些数据流模型并没有明显的同步操作（比如 Dryad^[8], DryadLINQ^[9], SCOPE^[10], FlumeJava^[11]）。当输入的数据准备好以后，下一个阶段就会开始。因为没有明确的同步操作，所以针对 MapReduce 等存在同步框架的优化技术对此通信框架没有任何效果。因此，很多研究人员针对采用没有显式阻塞的数据流通信模式的应用进行了特殊优化处理^[12,13]。

环形数据流模型。如图2.1 (d) 所示的是环形数据流通信模型。和无阻塞的数据流模型不同的是，在环形数据流中，数据流会重复使用之前阶段的部分组件。因为不同的阶段的数据流传输会复用组件，因此，可能会引起组件使用的冲突。Spark^[14] 通过在迭代中保留内存状态来消除循环冲突带来的开销。

2.2.1.3 大型并行数据传输模型 (Bulk Synchronous Parallel)

大型并行数据传输模型 (Bulk Synchronous Parallel, 简称 BSP) 是集群计算中的另一个众所周知的通信模型。使用这种模型的框架包括 Pregel^[15]、Giraph^② 和 Hama^③ 等，许多图形处理、矩阵计算等工具使用 BSP 框架。如图2.1(e) 所示，BSP 计算在一系列全局 superstep 中进行，每个 superstep 包含三个有序阶段：并行计算、进程通信和同步数据。在每一个 superstep 的末尾有明确的同步操作。

2.2.1.4 分区 – 聚合模型 (Partition-Aggregate)

图2.1(f) 显示是数据中心分区 – 聚合模型 (Partition-Aggregate)。分区 – 聚合模型是很多大型分布式计算通信模型。在分区 – 聚合模型中，请求发送到根节点，根节点把请求下发给底部的 worker 节点。随后，worker 节点把结果聚集到一起，然后把计算完成的结果，反馈给根节点。在数据中心，网络搜索、社交网络内容组成和广告选择等应用的通信模式均是基于此模式设计的。对于交互式的、实时的应

① Apache Hive. <http://hadoop.apache.org/hive/>.

② Apache Giraph. <http://incubator.apache.org/giraph/>.

③ Apache Hama. <http://hama.apache.org>.

用，用户得到响应的延迟是衡量服务质量好坏的重要指标。在减去传输等延迟之后，留给计算的时间通常只有 230~300 毫秒。

许多应用采用多层的分区－聚合模式，其中一个层的传输延迟会影响另一个层的启动。此外，响应请求可能需要迭代地调用，一个 aggregator 节点向下面的 worker 节点发出连续的请求，以准备响应（1 到 4 次响应是常见的，甚至有时能达到 20 次）。例如，在 web 搜索中，一个查询会发送给许多 aggregator 和 worker，每个组件负责不同的部分。根据回复，聚合器可能会细化查询并再次发送，以增加结果的完备性。因此，分区－聚合传输延迟过高会增加对查询结果性能的影响。

为了防止 SLA (Service Level Agreement) 性能被破坏，常常需要给 worker 节点设置一个截止期限 (deadline)，截止期限通常在 10ms~100ms。当一个 worker 节点错过了截止期限时，上层 aggregator 节点将会收到不完整的结果，这样会使的计算结果受到影响。因此，对于用户来说，越来越多的的流在截止期限之前完成对用户体验十分重要。例如，可能有 99.9% 的流在截止期限之前完成，意味着，1000 个请求中，有 1 个请求不能在截止时间之前完成。这样会严重影响用户的体验，造成客户的流失，造成不可逆转的损失。

2.2.2 网络流量特征分析

随着数据中心中部署应用的增多，对数据中心中流量特性的研究也逐渐增多。本部分对当前业界对数据中心流量测量的工作进行了总结和归纳。

查询流。查询流常常是由类似 web 这类请求产生的，这种请求的通信模型常常是 Partition-Aggregate。查询流非常短，并且对延迟十分敏感。当一个用户请求到达上层 aggregator 的节点时，上层的网络节点把请求下发给中层的节点，计算结果后，然后再下发给下层的 worker 节点。worker 节点得到请求后，读取请求的内容，然后计算结果，随后把结果反馈给根节点，根节点根据得到的反馈结果，进行处理。查询流的规模是非常有规律的，从高层的根节点到下层的 worker 节点的查询流的大小约为 1.6KB，从子节点回馈给根节点的流大小大约是 1.6KB 到 2KB。

背景流。背景流和查询流共同存在于数据中心。大多数背景流通常短于 1MB，但是数据中心超过 90% 的数据量来自约 1% 的背景流。这些长流通常在 1MB 到 50MB 之间。在分布式模型中，有很多拷贝和更新节点状态的背景数据流，这些流大小在 50KB 到 1MB。背景流的到达间隔时间反映了应用的不同服务的叠加和应用的多样性。背景流有如下的特性：（1）流到达时间间隔跨度很大，流的到达是服从重尾分布，在较短的时间间隔内，会有大量背景数据流到达。（2）长流发送是有一定周期性的，出现周期性是因为下层的叶节点会周期性的更新文件，此过

程中会产生大量流。

数据流并行程度和大小。在数据中心内，并行流数目的中位数为 36。99% 的机器之间连接数目在 1600 以内，有的机器连接的中位数，甚至在 1200^[16]。当只考虑长流 ($>1\text{MB}$ 时)，机器之间并发的长流数目较少，并行的长流的平均数为 1，并且大约 75% 的机器长流的并发数目在 2 以内。这些流，能持续几个 RTT，交换机的缓冲区大部分空间都是由这些流的数据包占据。

总之，对延迟敏感的查询流和需要高带宽的背景流同时存在于数据中心，在数据中心，这些数据流对网络资源的需求不同，我们需要设计合理的网络协议，来满足不同的流对网络资源的需求。

关于期限。Web 应用的交互性意味着延迟是决定服务质量的关键。研究表明，用户需要快速的回复响应^[17]。除去节点之间网络传输时间，应用通常具有 200~300ms 时间以完成其操作并对用户进行回复^[18]。

因此，诸如 Partition-Aggregate 这类模型的 SLA (Service Level Agreement) 要求 worker 等节点的询问和应答传输在截止期限 (deadline) 之前完成。任何没有在截止期限之前完成的数据流都被认定错失期限。错失期限的数据流越多，会有越多的上层计算节点获得不完整的结果，进而影响服务的质量。此外，在实际中不同的应用的数据流有不同的截止期限，甚至，同一个应用的不同阶段的传输的期限也

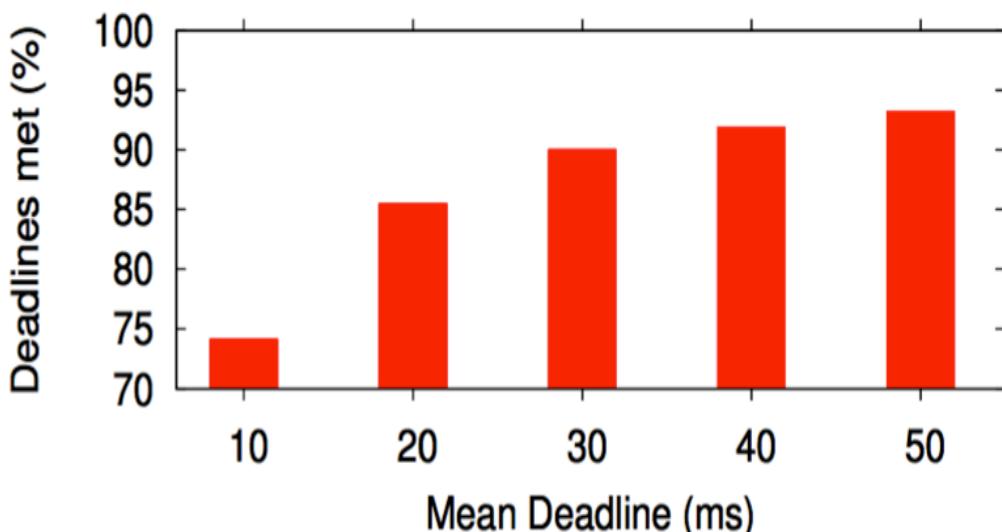


图 2.2 partition-aggregate 模型

不相同。在实际中，数据中心的流包括时间敏感的短消息（50KB 到 1MB），用于更新节点状态和传输时间长的背景流量（5KB 到 50MB）等。这些数据流在节点之间传递，不同业务的数据流具有不同的期限，有的甚至没有期限。图2.2显示的流传输期限分布情形，从中发现，超过 90% 的数据流的期限在 50ms 以内。事实上，数据中心很多在线服务都需要性能保障^[18,19]。用户的请求，需要满足延迟的需求。当请求到达时间节点时，无论计算结果如何，节点都需要把结果反馈给用户，如果错过了截止期限，应用的性能难以得到保障。

2.2.3 TCP 传输存在的问题

TCP 是一种面向连接的、可靠的、基于字节流的传输层通信协议。尽管 TCP 在 Internet 中被广泛应用，然而数据中心应用常常需要数据中心网络提供低延迟高带宽。因为数据中心网络资源有限，因此直接在数据中心使用 TCP 协议，会存在以下问题。

2.2.3.1 Incast 问题

如果很多数据流在短时间内收敛在交换机的同一接口上，数据包可能会耗尽交换机的缓冲区，导致数据包丢失，导致 Incast 问题^[20]。发生 Incast 常常是因为应用采取分区 – 聚合通信模式，因为在分区 – 聚合通信模式中，工作节点常常在较短时间间隔内会发送反馈给上层节点，导致上层节点交换机缓冲区溢出。

迄今为止的实验发现，Incast 经常会发生在实际生产环境中，Incast 会影响用户体验，从而影响企业的效益。这是因为，Incast 会导致交换机缓冲区过大，从而请求的数据流错过期限。如果使用 TCP 协议，则解决 Incast 问题的方法主要有：（1）为每个请求增加随机抖动时间。使用增加抖动时间的方法，可以避免 Incast 问题，但是，给每条流增加随机的抖动，会使一些短流增加额外的延迟，从而使数据流错过期限；（2）把数据流拆分成短流来适应交换机的缓冲区。使用拆分数据流的方法，会从语意上破坏应用的完整性。如果某条短流丢弃数据包，会使的整条长流传输不完整，从而破坏应用的性能，进而影响用户的体验。

2.2.3.2 交换机缓冲区队列过长

大多数数据中心的交换机是共享内存，交换机的所有端口共享交换机存储。到达交换机接口的数据包被存储到所有接口共享的高速存储器中。共享池中的内存由内存管理单元动态分配给数据包。内存管理单元尝试为每个接口分配尽可能多的内存，如果一个数据包必须排队等待出端口，但接口已经达到了最大内存分配

或者共享池本身已经耗尽，那么数据包就会被丢弃。构建大型多端口存储器非常昂贵，所以大多数便宜的交换机都是缓存都很小。

当前计算机网络判断拥塞的方法主要有三种：基于丢包的拥塞控制方法，基于 RTT 的拥塞控制方法，基于交换机队列长度的拥塞控制方法。TCP 使用基于丢包的拥塞控制方法，当网络中出现拥塞时，交换机的缓冲区会溢出，此时数据包会被丢弃。当发送端发现有丢包产生时，拥塞窗口会减半，从而减小发送端速率。如果不发生丢包，TCP 的滑动窗口会一直增大，直到占满端口缓冲区。因此使用 TCP 协议，会使的交换机缓冲区溢出并且交换机的队列过长，排队延迟高。数据中心的传输延迟往往 $< 1ms$ ，而排队延迟大约 $\sim 10ms^{[16]}$ ，因此排队延迟是数据中心主要延迟。使用 TCP 会使排队延迟高，造成用户体验差，进而影响应用性能和收益。因此在数据中心中，减小队列长度是减小延迟的有效方法。

2.2.3.3 公平分配带来的问题

数据中心并存了很多应用，这些应用对带宽和延迟的需求不同，使用 TCP 策略，带宽服从公平分配的原则。然而，不同的应用对延迟和带宽需求不同，使用 TCP，不能满足所有应用对带宽和延迟的需求。图2.3显示的是数据中心应用对带宽和延迟的需求，横轴表示的是延迟，纵轴表示的是带宽，在图2.3坐标系中，从左到右，延迟越来越大，从下到上越来越大。我们可以看到，对于 Hadoop 和数据

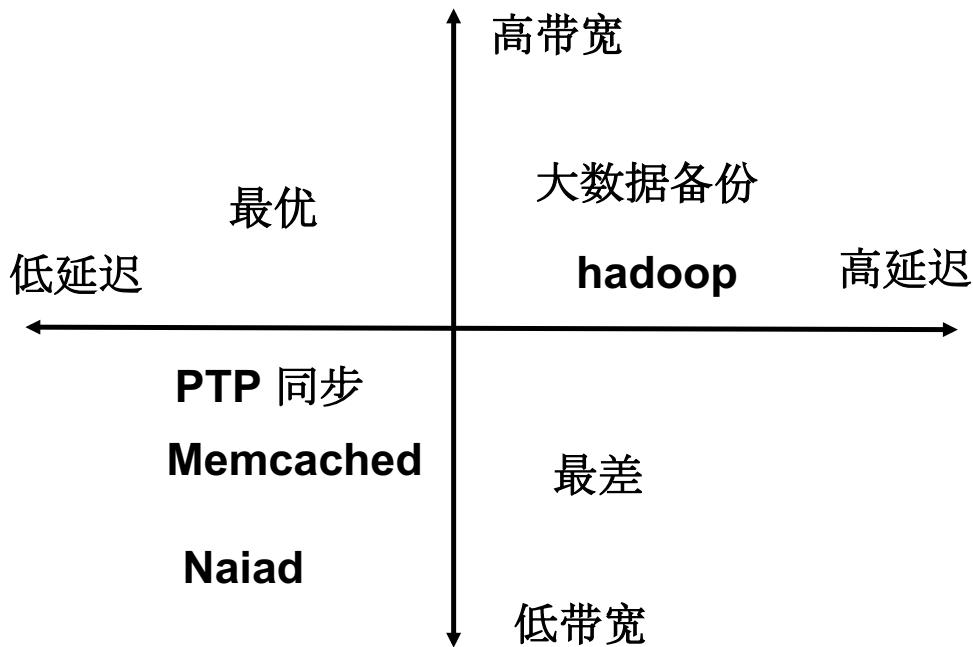


图 2.3 数据中心应用的需求

备份，需要高带宽，对延迟需求不大。对于 PTP 等应用，对带宽要求不高，但是需要低延迟。使用 TCP 协议，对这些应用同等看待，难以满足应用的不同需求，因此，亟需新的方法，考虑应用的不同需求，智能的分配网络资源，满足应用对网络资源的不同需求。

2.3 数据中心网络应用传输方案

日益使用的数据中心网络尽管有能给应用提供高带宽和低延迟的链路，但仍然面临频繁的网络拥塞^[21,22]。传统的 TCP 协议，难以满足不同的应用对带宽和延迟的需求，针对 TCP 在数据中心性能不足的问题，业界提出了一系列的方法来弥补。根据方法优化的粒度，可以分成流级别传输优化方案和任务级别传输优化方案。

2.3.1 流级别传输优化方案

流级别的调度方法，主要侧重提高截止期限前传输完成流的比例，以及优化流的平均完成时间。从方法上看，流级别的传输优化方法主要可以分成两类，第一种是通过改进传输协议 TCP 等，调整滑动窗口来调整带宽；第二种是进行传输调度，采用集中控制的方法，计算流在未来的一个时间段内的传输速率，使的流按照计算出的传输速率进行传输。本小节，我们从流的传输方法上对流传输进行总结和归类。

2.3.1.1 改进 TCP 的方案

在 DCN 中的许多改进 TCP 的传输方案中，DCTCP^[16] 提出在交换机的瞬时队列长度超过某个阈值 K 时，对数据包进行标记，随后，接收端对被标记的数据包的 ACK 进行标记，发送端通过统计 ACK 被标记的比例来计算拥塞程度，并根据拥塞程度 (α) 来调整滑动窗口。根据拥塞水平 α ，出现拥塞时，滑动窗口变化为 $w = w \times (1 - \alpha/2)$ 。通过设置适当的阈值，DCTCP 可以使的交换机的队列处于低水平上。大量的实验表明，DCTCP 有效地为短流提供低延迟服务。然而 DCTCP 不是根据截止期限进行带宽调整的，因此对于截止期限敏感的 OIDI (Online-Data-Intensive), MapReduce 等应用（如网络查询和广告），DCTCP 的调整效果会很差^[23]。

D^2TCP 改进了 DCTCP，能更加智能的调整发送端的拥塞窗口。它定义了一个期限逼近因子 $d = T/D$ ，其中 T 是不采用期限感知的方法，传输完成所需要的时间， D 是剩余时间。出现拥塞时，根据网络拥塞程度 α 和期限逼近因子 d 的拥塞窗口的调整方法： $w = w \times (1 - \alpha^d/2)$ 。 D^2TCP 不但具有 DCTCP 保持排队长度稳定并且

队列短的优良特性，而且可以使得期限更近的数据流获得更多的带宽。 D^2TCP 可以与 TCP 共存，并且易于在数据中心部署。

L^2DCT ^[24] 尝试减小流完成时间，和 D^2TCP 类似，当出现拥塞时， L^2DCT 也使用伽玛校正函数进行滑动窗口调整 ($cwnd = cwnd * (1 - \alpha^{w_c})$)。 L^2DCT 近似实现短流优先策略，小的数据流有更高的优先级，因此流平均完成时间变小。和 D^2TCP ， $DCTCP$ 一样， L^2DCT 也需要借助网络中 ECN 的标记。 L^2DCT 只能够优化流完成时间，而无法优化显式设置期限的数据流。

TIMELY^[25] 是基于 RTT 判断拥塞的协议，当 RTT 增大时，**TIMELY** 认为网络中发生了拥塞，发送速率需要降低。当 RTT 减小时，**TIMELY** 认为网络拥塞程度轻，可以增大发送速率。通过 RTT 进行拥塞判断，**TIMELY** 可以使的交换机队列处于低水平上，从而有效的降低传输延迟。但是，基于 RTT 的方法在数据中心存在以下两个问题：首先，数据中心网络中的 RTT 时间太短，无法准确测量；而使用 RDMA 这些新型技术的成本太高。此外，它不能和基于丢包的策略共存，因此在实际当中部署存在难度。

ICTCP^[26] 通过分析 TCP 带宽，RTT，和 TCP 接收窗口（receive window）的关系，动态的调整 TCP 接收端的接收窗口（receive window）大小，从而把网络中出现 Incast 的影响尽量降低。但是，数据中心 RTT 的波动大，难以进行测量；其次，**ICTCP** 侧重于优化网络中的 Incast 问题，无法对背景流进行传输优化。

2.3.1.2 流调度方案

D^3 ^[27] 在集中控制器上计算交换机上所需的带宽，并把计算的带宽反馈给交换机，使交换机按照计算出的速率发送。虽然 D^3 是第一个已知的基于截止期限进行带宽调整的传输协议，但是其对数据流优先级调度采用贪心和先到先得的策略，因此在某些情况下，期限远的数据流会比期限近的数据流优先调度^[27]。另一方面， D^3 需要修改交换机硬件，因此在实际中部署难度大。

PDQ^[28] 在端上使用抢占式流量调度来分配带宽。与 D^3 不同，**PDQ** 以分布式方式工作，其抢占式调度可以处理 D^3 无法处理的场景。**PDQ** 可以使用不同的调度策略，如最早截止期限（Earliest Deadline First，简称 EDF）或最短作业优先（Shortest Job First，简称 SJF）。然而，SJF 是单一链路上的流平均完成时间最小化的最佳解决方案。对于复杂链路来说，它并不是最优的。**PDQ** 使用流平均完成时间被用作评估指标，**PDQ** 还要求修改交换机的硬件，不能与传统的 TCP 共存。

PIAS^[29] 借助于交换机当中存在的队列，实现了 MLFQ（Multiple Level Feedback Queue）机制。在这种机制中，流的优先级开始最高，随着数据发送，流的优先级

开始降低，PIAS 近似实现了短流优先的策略，从而间接的优化了流平均完成时间。然而 PIAS 只是优化流平均完成时间，并没有考虑网络拥塞情形，当网络拥塞时，并不能满足紧急应用的数据流对带宽的需求。

QJUMP^[30] 给每个应用分配一个优先级，对高优先级的应用在发送端进行限速，而一旦数据包进入网络中，高优先级的数据包可以排在交换机的前列，低优先级的数据包排在交换机队列的后面。从而实现了高优先级的数据包有较低的延迟，而低优先级的数据包，有更高的网络带宽。**QJUMP** 在高带宽和低延迟找到了一个平衡，但是，在实际应用中，应该合理的给应用分配优先级，如果优先级分配不合理，会导致应用性能降低。

在 **FastPass**^[31] 中，发送端何时发送数据包，以及发送数据包的路径由中央处理器的计算结果决定。**FastPass** 借助数据包发送时间算法和数据包发送路径算法来决定数据包发送时间和发送路径。通过这两个算法，**FastPass** 能够在不改变硬件的前提下，使交换机队列浅，从而降低排队延迟。**FastPass** 可以有效的降低网络中交换机队列长度，但是 **FastPass** 是集中式控制的方法，集中式控制的方法在数据中心中会增加传输延迟。此外，如果控制器发生故障，会导致系统崩溃。

pFabric^[32] 根据流剩余数据量大小进行调度，**pFabric**^[32] 提出的分布式算法近似于最短作业优先（Shortest Job First，简称 SJF）策略，并且被证明在优化流平均完成时间问题上是近似最优策略。与 PDQ 相比，**pFabric** 将流量调度与速率控制分离开来，实现起来更加简单，并且可以与类似 TCP 的方案共存。**pFabric** 需要单独的交换机，这导致 **pFabric** 在网络中难以部署。

DeTail^[33]，侧重于跨层技术，包括流的优先级调度和高效的负载平衡策略，以减少长流的完成时间的。尽管能有效的减少网络传输延迟，但是 **DeTail** 需要使用新的网络协议，因此在实际难以大规模使用。

Karuna^[34] 是调度混合流（有期限的流和没有期限的流混合）场景下的解决方案。**Karuna** 主张，对于有期限的流，要在截止期限之前完成，对于没有截止期限的流，平均流完成时间影响最小。**Karuna**^[34] 可以减少流失时间的流量百分比，并减少流的平均完成时间。**Karuna** 必须在交换和终端节点上进行改变。这使得 **Karuna** 在实践中部署难度很大。

PASE^[35] 声明的协议（DCTCP，PDQ，**pFabric** 等）不是竞争关系，它们是互补的。**PASE** 不改变网络中的器件，在减少流完成时间方面性能要比 **pFabric** 复杂。

2.3.2 任务级别传输优化方案

流级别的传输优化方案，能够优化单条流的完成时间，或者尽量保证流在截止期限之前完成。但是，单纯的流级别传输并不能满足应用的需求。比如 web 搜索（分区-聚合模型），数据备份，MapReduce，分布式存储等应用是包含很多并行的数据流。对多阶段的应用，只有当上一阶段传输完毕后，下一阶段才能开始传输，因此，为了应用能够取得更好的性能，需要从流传输任务的级别进行优化。

Orchestra^[6] 发现任务传输时间占据了任务总处理时间的 33%^[6]。因此，Orchestra 首先提出调度应该侧重优化任务阶段整体传输，而不只是优化某条数据流。Orchestra 使用一个全局的控制器来对所有传输信息进行同步，这个全局控制器存储了所有源和目的节点信息。对于广播的传输网络传输结构，Orchestra 采用 Cornet 协议，更加合理的利用网络拓扑进行传输优化。对于 shuffle 的传输，Orchestra 采用权重 shuffle 调度进行带宽分配。Orchestra 从总体上优化了任务传输，尤其对广播和 MapReduce 的 shuffle 阶段性能提升显著。

Barrat^[36] 提出在调度应用的数据传输时，应该在任务的级别进行，而不是在流级别进行。Barrat 对任务的调度整体采用 FIFO 模式，对于数据量大的任务，随着传输进行，降低任务优先级，从而保证大任务不会阻塞链路，影响小任务的传输。Barrat 是一个分布式的任务级别调度系统，通过对任务的整体调度，Barrat 能够有效的减小任务平均完成时间，同时减少任务最长的传输延迟，进而提升应用的传输性能。

Varys^[37] 提出对于存在数据并行的应用，应该以 coflow^[38] 为调度单元，而仅仅在流级别进行优化是不够的。Varys 提出对应用的优化的目标是最小化平均流组（coflow）完成时间。为了达到这个目标，Varys 采用集中式调度策略，首先计算流组的优先级，然后计算每条数据流的带宽。对于流组优先级，Varys 采用最小瓶颈优先（Smallest-Effective-Bottleneck-First，简称 SEBF）启发式算法来决定。最小瓶颈优先策略策略是把一个流组（coflow）中传输最慢的流的完成时间当做流组的预估完成时间，然后根据预估传输时间进行排序，使得预估完成时间小的流组获得高优先级。对于数据流的带宽的控制，使用最小带宽期望（Minimum-Allocation-for-Desired-Duration，简称 MADD）算法。使用集中控制的方法，借助以上两个策略，Varys 优化了平均流组完成时间（Coflow Completion Time，简称 CCT），从而提高了应用的性能。

Varys 需要预先得知很多流组（coflow）的信息，事实上，在数据中心中，很多应用的流组（coflow）中流的大小信息是很难预先得知的，尤其在 hadoop 多阶段计算中，中间 shuffle 会传输大量数据，数据流的大小很难被预先得知。基于此，

Aalo^[39] 提出，在调度 coflow 时，应假设预先不知流组（coflow）中流的大小进行调度。Aalo 实现了最短流组优先策略，根据已经发送数据量的大小，把流组（coflow）分配到不同的优先级队列中。相同的优先级队列中的流组（coflow）采取 FIFO 调度机制。Aalo 近似实现最短流组（coflow）优先的策略，整体上调度了流组（coflow），从而优化了流组（coflow）的传输效率。

事实上，无论 Aalo 还是 Varys，都需要提前分别出应用的流组（coflow），这需要对应用进行改动。而修正比如 Hadoop 和 Spark 等应用的 API 需要考虑兼容性和进行 Java-byte 级别的改动，这导致修改的复杂度较高。CODA^[40] 首先尝试不改变应用，来自动的区分和调度应用流组（coflow）。同时使用 late binding 等措施来尽量减少流组（coflow）区分的错误等。通过对应用的流组（coflow）进行有效的区分和带宽的调度，CODA 从整体上对 coflow 实现了调度，从而，有效的降低了应用流组（coflow）的传输时间，优化了应用传输延迟。

和 Varys，以及 Aalo 等相同，D-CAS^[41] 将数据网络假设成非阻塞交换机，认为传输冲突只发生在交换机的入端口以及出端口，中间路径并没有拥塞发生。D-CAS 提出 coflow 最小化平均传输完成时间调度问题和开放商店中任务平均工作完成时间问题的优化等价。为了优化 coflow 传输延迟，D-CAS 改进了开放商店中的任务平均完成时间的 2-近似算法^[42]，并将 2-近似调度算法引入到流组（coflow）的调度中，然后改进 2-近似调度策略，从而得到流组（coflow）的在线调度算法。D-CAS 采用分布式的方法实现，可以有效的减小流组（coflow）的传输时间。

RAPIER^[43] 打破了数据中心非阻塞体系结构，认为在传输中，拥塞随时发生，不只发生在第一跳和最后一跳。传输时不应只考虑调度问题，为每条数据流分配合理的路径是很重要的问题。基于此，RAPIER 把流组（coflow）的路径选择和流组（coflow）的调度放在一起考虑，通过求解非整数优化问题，把每条数据流发送速率的大小和发送路径都计算出来。RAPIER 可以减少平均 CCT 在 50% 以上，并且易于在真实环境中部署。和 Varys 相同，RAPIER 需要预先得知 coflow 中每条数据流大小等信息。

Task-aware TCP^[44] 认为传统的 TCP-based 方法只考虑了单一的情形，而没有考虑流之间的关系，这会影响流之间的传输，导致传输效率低。Task-aware TCP 主张，在一个 task 中，速率过快的数据流应该把带宽让给速率慢的数据流。Task-aware TCP^[44] 改进了 DCTCP 方法，借助于 ECN 标记，区分出传输过快和传输过慢的数据流，把传输过快的数据流带宽给传输过慢的数据流一些，从而使的任务的平均传输时间减少，进而优化了任务的传输延迟。

Qiu 在^[45] 从理论角度考虑有权重的流组（coflow）的调度，假设已经知道 coflow

到达时间，每条流的信息进行调度。Qiu 同时强调调度流组（coflow）时，应该也同时考虑权重，而不应该把所有流组（coflow）同等看待。使用近似算法，Qiu 提出的策略，可以达到 $\frac{67}{3}$ 的近似比。使用随机性策略，算法的近似度可以达到 $9 + 16\sqrt{2}$ 。Qiu 在^[45] 通过求解方程组对流组（coflow）进行调度求解，此方法有较高的复杂度，在实际中难以部署。

2.3.3 数据中心传输方案存在的问题

相比于传统的 TCP，当前数据中心传输方案能够有效的提高数据中心传输效率，提高应用对网络等资源的使用情况，然而，尽管数据中心网络技术在不断发展，但是，随着应用业务的不断发展，数据中心网络始终难以满足应用的各种需求，因此，需要更加合理的网络资源调度来满足应用对网络资源的需求。当前数据中心网络依然存在以下的问题：

(1) 未考虑网络拥塞程度和应用传输期限关系

D^3 首先提出，在进行传输时，应该给数据流分配足够多带宽，让数据流满足期限。 D^3 改动交换机，并在交换机上计算出数据流传输满足期限的最小带宽，从而数据流能够获得足够的带宽，从而在期限之前完成。 D^2TCP 根据网络的拥塞情形和数据流期限调整数发送窗口，通过拥塞窗口间接的影响数据流传输的带宽，进而实现不同需求的数据流达到各自传输的目的。这两种方法都同时考虑了数据流传输的期限和网络拥塞状况。然而，无论哪种方法，都没有考虑网络拥塞程度，期限的二维关系。事实上，尽管 D^2TCP 考虑了数据流的传输应该同时考虑网络期限，但是当网络拥塞程度严重时， D^2TCP 不再是基于期限调整的拥塞控制策略。因此，如何合理的根据网络拥塞情形，期限调整数据流传输是亟需解决的问题。

(2) 无法同时满足延迟敏感流和带宽敏感流的需求

数据中心中同时存在对延迟敏感的短流和对带宽敏感的背景流。当前的数据中心的传输方案，无论是 D^2TCP , D^3 还是 L^2DCT 都是解决单一数据流的需求，即或者满足延迟敏感流的需求，或者满足带宽敏感流的需求，比如 D^2TCP , D^3 侧重优化有显式期限的流， L^2DCT 优化的是流平均完成时间，并不能同时满足这两种数据流的需求。

(3) 任务级别调度只考虑网络拥塞并没有考虑任务的重要性

尽管 Varys, Aalo 等策略在调度数据中心资源时，是在任务级别进行的，但是这些策略只是考虑网络的拥塞状况，目标是优化应用的传输完成时间。然而，应用有不同的语意，只考虑应用的传输是不够的，应该同时兼顾应用的重要性。比如，进行计算的 hadoop 和视频传输业务共享数据中心资源，进行计算的 hadoop 的业

务的优先级比进行视频传输业务的优先级高，使用传统的传输策略，会认为这两个应用数据流的优先级相同，这样常会导致进行计算的 Hadoop 业务的数据流得到的带宽不足，造成 Hadoop 的完成时间过长，从而影响应用的整体性能并且造成网络资源利用率偏低。

基于以上的几个问题，本文从流级别和任务级别对传输策略进行优化和改进，使的流的带宽能够根据网络拥塞和流的期限进行动态调整，同时对混合场景下的流传输进行优化，使有期限的数据流和对带宽敏感的数据流都能满足需求。最后从任务级别对传输进行了优化。

2.4 本章小结

本章中，我们综述了数据中心应用传输方案的相关工作。我们从常见应用的通信模型和数据流量等方面对数据中心传输进行了概述。然后，我们总结了流级别和任务级别的数据中心传输方案进行。最后，我们总结了数据中心传输方案存在的问题。

第3章 基于ECN标记的流传输模型

3.1 本章引言

本章中，我们介绍基于 ECN 标记的流模型。首先，我们对基于 ECN 标记的流模型的场景进行介绍。然后，我们介绍基于 ECN 标记的流模型，最后，我们根据基于 ECN 标记的流模型实例化 DCTCP，得到 DCTCP 流模型，作为 ECN 标记的流模型的一个实例。

3.2 场景描述

传统 TCP 通过调整拥塞窗口来进行速率的控制，当网络不拥塞时，拥塞窗口每个 RTT 增加 1 MSS，当网络中出现拥塞时，TCP 拥塞窗口减半。通过对 TCP 滑动窗口的调整，TCP 实现了对发送速率的控制。

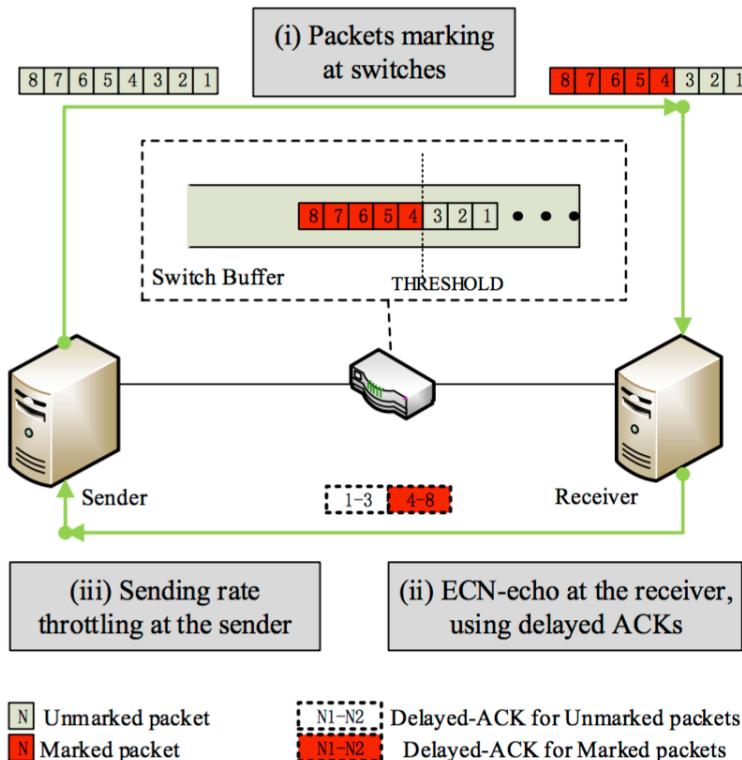


图 3.1 基于 ECN 标记的数据中心速率控制过程，交换机上设置阈值，当排队队列超过阈值后，进行标记，发送端根据被标记的数据包比例，动态调整发送窗口

基于 ECN 标记的拥塞控制速率机制过程如图3.1所示：首先，发送端发送数据

包给交换机，在交换机设置一个阈值，当交换机队列长度超过阈值后，超过交换机设置队列设置阈值的数据包会被标记 CE（和 TCP 不同，TCP 会丢包）。当接受端收到数据包后，检查数据包是否被标记 CE，如果发送的数据包被标记 CE，那么给发送端回复的 ACK 中标记 ECN。发送端检查收到的 ACK 中被标记 ECN 的 ACK 的比例，以此来计算网络的拥塞程度 α ，根据拥塞程度进行拥塞窗口的计算。

当网络中没有拥塞时（此 RTT 时间内没有被标记的数据包），滑动窗口每个 RTT 增加 $(1 - f_1(\alpha))$ ，当网络中有拥塞时（此 RTT 时间内中有被标记的数据包），滑动窗口减小 $f_2(\alpha)$ 倍。其中 $f_1(\alpha)$ 和 $f_2(\alpha)$ 是关于拥塞程度 α 的函数。 $(3-1)$ 是基于 ECN 标记的拥塞避免阶段窗口变化的公式。

$$w = \begin{cases} w + (1 - f_1(\alpha)) & \text{没有拥塞} \\ w \times (1 - f_2(\alpha)) & \text{出现拥塞} \end{cases} \quad (3-1)$$

3.3 ECN 标记流模型

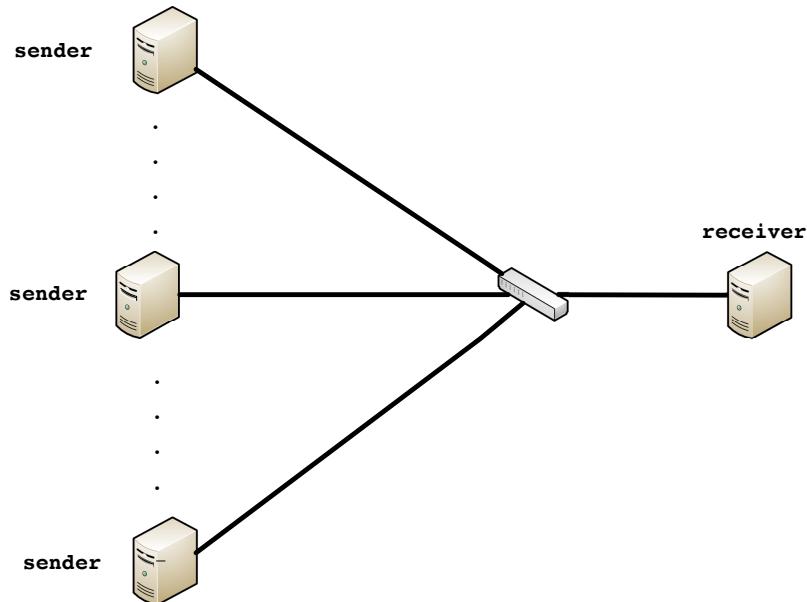


图 3.2 N 个发送端发送数据给 1 个接收端

如图3.2所示，假设 N 个发送端连接 1 个交换机，1 个接收端连接在交换机上。N 个发送端使用公式 $(3-1)$ 进行拥塞控制窗口的计算。交换机支持 ECN 标记，当

交换机队列满足 ECN 标记条件时，发送到交换机的数据包会进行数据包标记。

3.3.1 模型描述

公式 (3-2)~(3-4) 是基于 ECN 标记的描述模型，其中 (3-2) 表示的是窗口变化过程，其中，第一部分 $\frac{1-f_1(\alpha)}{R(t)}$ 描述的是窗口增加的过程，其中每个 RTT 窗口增加 $1 - f_1(\alpha)$ 。 $\frac{w_i(t)f_2(\alpha)}{R(t)}p(t - R^*)$ 描述的是窗口减小的过程，当发生拥塞时，窗口减小 $\frac{w_i(t)f_2(\alpha)}{R(t)}$ 。其中 $p(t - R^*)$ 表示的是是否出现 ECN 标记，当 $p(t - R^*) = 1$ 时，表示 ACK 中有 ECN 标记，当 $p(t - R^*) = 0$ 时，表示 ACK 中没有 ECN 标记，此时网络中没有拥塞。

公式 (3-3) 表示的是交换机队列变化， $\sum_{i=1}^N \frac{w_i(t)}{R(t)}$ 表示的是发送到交换机的数据包，假设交换机的转发速率是 C pkt/s，那么，交换机的队列长度是 $\sum_{i=1}^N \frac{w_i(t)}{R(t)} - C$ 。

公式 (3-4) 是交换机是否进行标记，当交换机队列满足一定条件 $F(q(t)) == 1$ 时，进行标记的函数 $\hat{p}(t) = 1$ ，否则， $\hat{p}(t) = 0$ 。

$$\frac{dw_i}{dt} = \frac{1 - f_1(\alpha)}{R(t)} - \frac{w_i(t)f_2(\alpha)}{R(t)}p(t - R^*) \quad (3-2)$$

$$\frac{dq}{dt} = \sum_{i=1}^N \frac{w_i(t)}{R(t)} - C \quad (3-3)$$

$$\hat{p}(t) = 1_{F(q(t)) == 1} \quad (3-4)$$

3.3.2 模型实例

本章节，我们使用 DCTCP^[16] 来验证模型，DCTCP 协议在交换机上设置阈值 K，当交换机的队列超过 K 时，数据包被打标记，同时在发送端滑动平均拥塞程度 α ：

$$\alpha = (1 - g) \times \alpha + g \times F \quad (3-5)$$

其中，g 为滑动平均因子，F 为当前 RTT 中被标记 ECN 的数据包，当前滑动平均拥塞程度 α 为上一个 RTT 的打标记的 ECN 比例 F 和当前计算的平均拥塞程度的比例值。在 DCTCP 中， $f_1(\alpha) = 0$ ， $f_2(\alpha) = \frac{\alpha}{2}$ ，将 $f_1(\alpha)$ 和 $f_2(\alpha)$ 带入 (3-1) 得到拥塞控制函数：

$$w = \begin{cases} w + 1 & \text{没有拥塞} \\ w \times (1 - \frac{\alpha}{2}) & \text{出现拥塞} \end{cases} \quad (3-6)$$

同时把 $f_1(\alpha)$ 和 $f_2(\alpha)$ 带入 (3-2)~(3-4)，并且改动 (3-5)，得到：

$$\frac{dw_i}{dt} = \frac{1}{R(t)} - \frac{w_i(t)\alpha}{2R(t)} p(t - R^*) \quad (3-7)$$

$$\frac{dq}{dt} = \sum_{i=1}^N \frac{w_i(t)}{R(t)} - C \quad (3-8)$$

$$\frac{d\alpha_i}{dt} = \frac{g}{R(t)} (p(t - R^*) - \alpha_i(t)) \quad (3-9)$$

$$\hat{p}(t) = 1_{q(t) > K} \quad (3-10)$$

其中，(3-7) 是描述的是 DCTCP 的窗口，(3-8) 描述的是队列长度，当没有出现拥塞时，拥塞窗口每个 RTT 增加 1，当出现拥塞时，滑动拥塞窗口减小 $\frac{\alpha}{2}$ 。特别的，(3-9) 是描述的拥塞程度 α ，(3-10) 中被标记的条件是队列长度大于 K 。设置 $g=1/16$, $C = 10\text{Gbps}$, $N = 2$, $K = 64$, 图3.3是对 DCTCP 实例化的模型进行的验证并且使用 ns-2 和模型结果进行对比的结果。

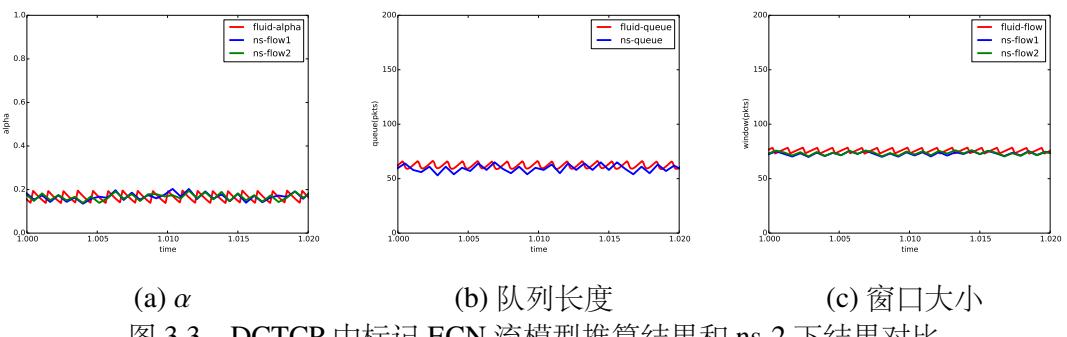


图 3.3 DCTCP 中标记 ECN 流模型推算结果和 ns-2 下结果对比

图3.3(a) 描述的是拥塞程度 α ，图3.3(b) 描述的是队列长度，图3.3(c) 描述的是拥塞窗口，我们可以发现，通过模型进行描述推倒的结果和仿真的结果基本相同。因此，使用模型可以有效的描述基于 ECN 标记的传输协议。

3.4 本章小结

本章中，我们介绍了基于 ECN 标记的流模型。我们介绍了基于 ECN 标记的流模型的场景和基于 ECN 标记的流模型。最后，我们根据基于 ECN 标记的流模型实例化 DCTCP，得到 DCTCP 流模型，作为 ECN 标记的流模型的一个实例。

第4章 基于负载自适应原则的传输优化方案

现在很多在线应用部署在数据中心，因此数据中心网络的性能和底层技术吸引了越来越多的研究者的兴趣。为了达到更好的网络性能，近期的研究从不同角度优化数据中心网络流量传输和调度，并且设计各种路由和传输方案。特别是对于必须及时为用户服务的应用，数据流的传输应尽快的完成，有的必须在截止时间之前完成，最近业界出现了一系列的考虑流传输期限的网络流量控制策略。在本章中，我们主张在设计数据流传输方案时，应该遵循一个简单的原则：拥有不同截止期限（deadline）的数据流在带宽分配和占用上应该被区分开，网络负载越重，数据流越应该被区分。我们认为在设计流量控制策略时应该遵循这个原则。

根据这个原则，我们并提出了一种简单的拥塞控制算法 – 正比负载差分策略（Load Proportional Differentiation，简称 LPD）作为其应用。我们已经在不同的拓扑和负载情况下评估了 LPD。我们发现与 D²TCP（最先进的基于期限拥塞控制滑动窗口策略）相比，使用 LPD 错过最后期限的数据流的比例减少 25% 以上。与 Karuna 相比，LPD 平均有 5% 的性能损失。但是在拥塞严重的情况下，LPD 性能比 Karuna 高 5% ~10%。事实上“越拥塞，越区分”是一个通用的原则，它也可以用于其他目标的优化。我们考虑优化流平均完成时间。与基于窗口的协议 L²DCT 相比，LPD 可以将流平均完成时间减少 30%。和最先进的流平均完成时间优化策略 pFabric 相比，LPD 只有 20% 的性能损失。

4.1 概述

目前，越来越多的服务和应用，如网络搜索和社交网络被托管在数据中心^[34,37~39,46]。由于数据中心网络的性能，特别是数据传输时延，吞吐量等直接影响到用户获得的服务的质量，因此需要设计新的网络协议，充分利用数据中心提供的网络资源。在数据中心网络（DCN）中，短流需要低延迟，长流需要高吞吐量^[16,34]。为了满足这些要求，最近的研究提出了各种各样的为数据中心量身定做的流量调度策略。

特别是，在数据中心普遍部署了一类称为在线数据密集型（Online Data Intensive，简称 OLDI）^[23,47] 的重要应用。这些必须及时的为用户提供服务。即使应用增加 1 毫秒的延迟也会对企业利润产生影响^[4]。这些应用通常需要查询和收集来自节点的反馈结果。为了满足应用的响应期限，即使某些应用的计算结果尚未计算完毕，应用程序也可能向用户发送不完整的反馈结果。这进一步要求应用程序数

据传输在截止时间之前完成，因为错过期限的数据流越少，服务质量越高，用户的体验越好。像 TCP 或 DCTCP 这类没有考虑期限的传输方案在这些应用中效果不佳，而很多现代数据中心流量控制算法^[23,27] 或流量调度算法^[28] 中明确地考虑了截止日期。

在本章中，我们重点研究数据中心基于期限的流量控制算法。我们基于 D²TCP^[16] 开始我们的工作。通过仔细研究不同业务负载情况下的 D²TCP 性能，我们注意到，当网络负载很高的情形下，D²TCP 退化为 DCTCP，并且不再是基于期限的拥塞控制策略。考虑到这一点，我们认为当网络负载较重时，应该为截止期限较近的流分配较多的网络带宽，并提出一个简单的期限感知速率控制原则：拥有不同的期限的数据流在带宽分配上应该有区分，并且网络负载越重，区分程度越大。

为了证明这个原理的有效性，我们还设计了一个简单的基于期限的拥塞速率控制算法，正比负载差分策略（Load Proportional Differentiation, 简称 LPD）作为其应用。LPD 的基本思想是：使用截止期限来调节拥塞窗口时，网络负载应作为调整因子引入其中，数据流的调整程度与网络负载成正比。我们使用典型的数据中心拓扑和各种业务负载场景来评估 LPD 的性能，并将其与一些最新的数据中心流量控制算法（例如 DCTCP^[16]，D²TCP^[23] 和 L²DCT^[24]，Karuna^[34]）进行对比。实验结果显示，LPD 性能总是能超过这些策略，LPD 能使更多流在截止期限之前完成。与 D²TCP 相比，LPD 可以将错过最后期限的流的比例减少 25% 以上。和最先进的基于期限的拥塞控制方法 Karuna 相比，LPD 的性能平均差 5%。但对于一些较为拥挤的场景，由于 LPD 采用“越拥塞，越区分”的设计原则，在拥塞程度严重的时候，LPD 的性能比 Karuna 好 10%。此外，“越拥塞，越区分”是一般性的原则，它可以用来优化其他目标，诸如减少流平均完成时间。本文修改 LPD 以使其适合于优化流平均完成时间，然后将其性能与最先进的基于窗口的优化流平均完成时间的协议 L²DCT 进行比较，发现 LPD 比 L²DCT 在流平均完成时间方面缩短 30%。即使与最先进的优化流平均完成时间的方法 pFabric^[32] 相比，LPD 的性能差 20%。本文随后在 ns2^[48,49] 上和 Linux kernel 3.2.61^[50] 中实现了 LPD。

本章的主要工作如下：

- (1) 测试并发现当前 DCNs 中基于期限的拥塞控制策略在网络重度拥塞严重的情形下失效。
- (2) 提出了“越拥塞，越区分”的原则来设计 DCNs 中的基于截止期限流量控制机制，并在数学上确定了遵循这一原则的方案的充分和必要条件。
- (3) 提出 LPD 算法作为“越拥塞，越区分”原则的一个应用，并建立基于 ECN

的数据流传输模型来分析它的性能。

(4) 在 ns-2 和 linux 内核 3.2.61 中实现 LPD，并且把 LPD 的性能和当前最流行的基于期限的速率控制策略进行对比。

LPD 的模拟代码可以在^[49] 下载，LPD 的 Linux 内核源代码可以在^[50] 中找到。本文还根据“越拥塞，越区分”的原则设计了其他的拥塞控制策略，并且说明此策略是具有普遍性的。

4.2 相关工作和研究动机

4.2.1 相关工作

表4.1表示的是流级别的各类传输方案。可以发现，通过流量控制来优化传输

表 4.1 流级别优化方案

时间	流控	调度
2010	DCTCP _[sigcomm]	
2011		D ³ _[sigcomm]
2012	D ² TCP _[sigcomm]	PDQ _[sigcomm]
2013	L ² DCT _[Infocom] , MPTCP _[Hotnets]	pFabric _[sigcomm]
2014	TIMELY _[sigcomm]	
2015		PASE _[NSDI] , QJUMP _[NSDI] , FastPass _[sigcomm]
2016	Karuna _[sigcomm]	

延迟的方案有 DCTCP^[16], D²TCP^[23], TIMELY^[25], Karuna^[34], MPTCP^[51]。通过流量调度的方案有 D³^[27], PDQ^[28], pFabric^[32], PASE^[35], QJUMP^[30], FastPass^[31]。

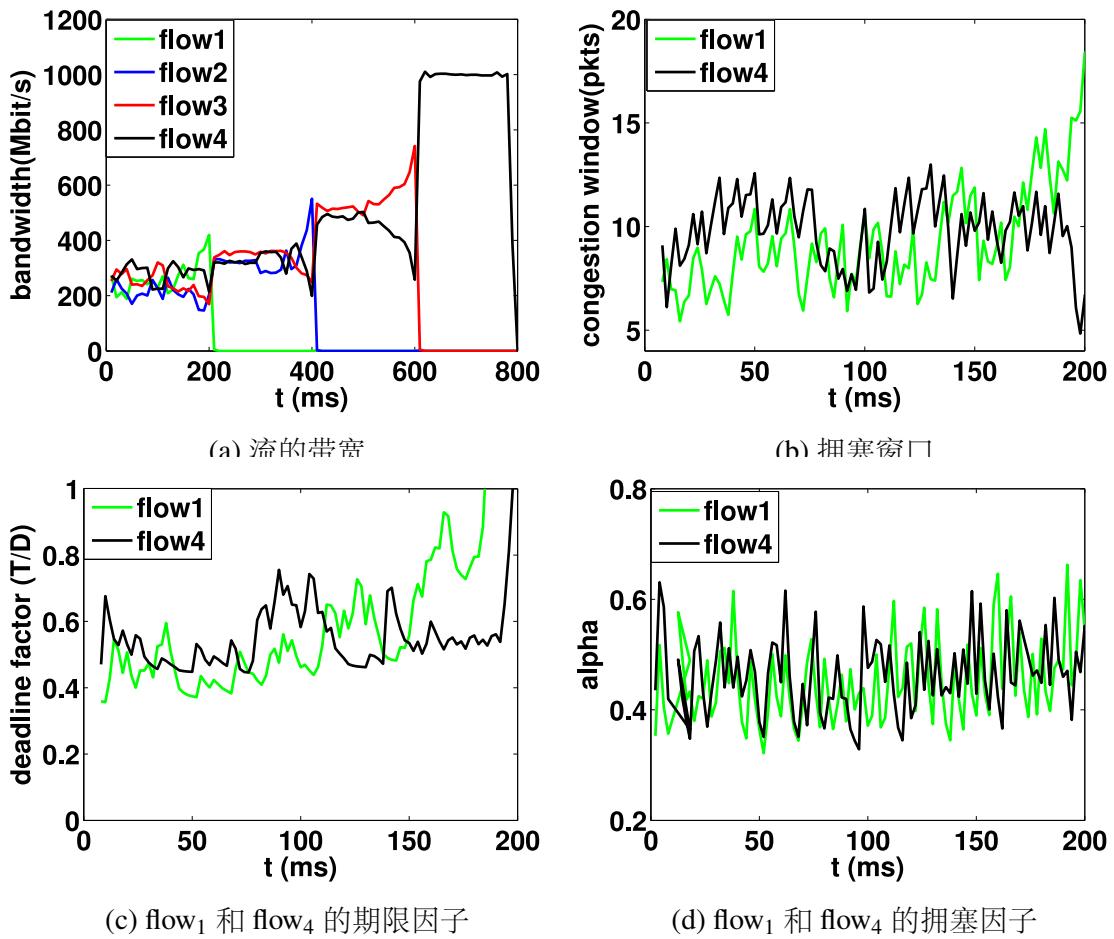
4.2.2 研究动机

D²TCP 给不同的数据流分配不同大小的带宽，使期限近的流获得更多带宽。本文认为这是一个重要的尝试，但是在拥塞控制中仅仅考虑期限是不够的，网络拥塞也应被考虑。

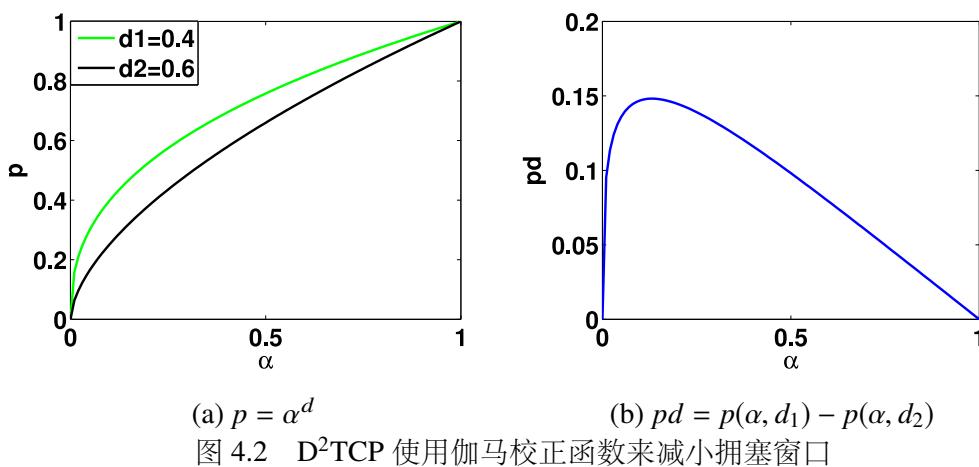
为了验证 D²TCP 存在的问题，本部分使用 ns-2 进行如下仿真：假设四条流通过瓶颈带宽为 1Gbps 链路，流同时启动，大小分别为 8,16,32 和 40 MB，截止期限分别为 200,400,600 和 800 ms，设置 RTT 为 100 us。如果使用 TCP，那么可以

表 4.2 期限之后，未传输完的数据量

	D ² TCP	DCTCP
flow 1 (8 MB, 200 ms)	1.53 MB	2.07 MB
flow 2 (16 MB, 400 ms)	1.79 MB	2.14 MB
flow 3 (32 MB, 600 ms)	3.64 MB	4.08 MB
flow 4 (40 MB, 800 ms)	0 MB	0 MB

图 4.1 D^2 TCP 的性能：4 条并发流

使得所有流在最晚的截止期限 (800ms) 之前完成。流传输结果如表4.2第四列所示，表4.2中每个数字表示在流截止期限到达时剩余多少数据未被发送完成（传输完成后，数据流会被强制停止）。把非基于期限调整的策略 DCTCP 也放在最后进行比较。从表4.2可以看到，尽管 D^2 TCP 在流截止时间 (deadline) 之前传输的数据比 DCTCP 多，但不能有效地减少错失期限流的数目。在这两种情况下，只有 flow4 在截止时间之前完成。根据实验结果显示，在一些场景下， D^2 TCP 的性能和 DCTCP



的性能类似。

图4.1显示了D²TCP传输的更多细节，D²TCP使用 $w = w \times (1 - \alpha^d / 2)$ 来调整其拥塞窗口 w ，其中 α 是估计的拥塞程度， d 是期限逼近因子。图4.1 (a) 描绘了 flow₁ 和 flow₄ 的带宽。可以发现，大部分时间流之间带宽差别很小。图4.1 (b) 是 flow₁ 和 flow₄ 在最初 200ms 期间的拥塞窗口大小，除了快接近最后的 10ms 时，flow₁ 的窗口会比 flow₄ 大很多之外，大部分时间，flow₁ 和 flow₄ 的差距不大。但是，图4.1 (c) 所示，flow₁ 和 flow₄ 的期限逼近因子相差很大，特别是在 100 ms 到 200 ms 的时间内。为了完整性，图4.1 (d) 也绘制了 α ，它表示 flow₁ 和 flow₄ 对拥塞程度的一种评估。

4.3 “越拥塞，越区分”的设计原则

本章的工作主要借鉴 DCN 中较早的速率控制方案，如 DCTCP, D²TCP 和 L²DCT 等。特别是 D²TCP，通过将期限控制函数与 DCTCP 相结合，本质上给不同的数据流分配不同的带宽，从而相同条件下期限紧迫的流获得更多的带宽。正如4.2.2节所示，本文认为仅仅通过期限对数据流进行带宽调整是不够的。在4.3.1节中，本文分析出现这种不好的情况的根本原因。在4.3.2节，本文提出基于截止期限的速率控制策略的设计原则，并且使用数学语言对之进行描述。

4.3.1 分析伽马校正函数

本部分对 D²TCP 进行深入探究。D²TCP 的核心思想是在惩罚函数中使用伽马校正函数 $p(\alpha, d) = \alpha^d$, 其中 $0 < \alpha < 1$ 。图 4.2(a) 显示的是 $d = 0.4$ 和 $d = 0.6$ 时伽马校正函数的曲线。可以发现, 当 α 增加时, p 也增加, 因此可以认为网络负载加重, 拥塞窗口变小。这个性质对于基于期限的速率控制确实是合理的和必要的。然而,

如果考虑惩罚函数的差异，例如 $pd(\alpha, d_1, d_2) = p(\alpha, d_1) - p(\alpha, d_2) = \alpha^{d_1} - \alpha^{d_2}$ ，其中 $d_1 < d_2$ ，可以发现 pd 首先随 α 的增加而增加，在 α 经过某一点（图4.2 (b) 中 0.2 左右）之后减小。例如， $pd(0.4, 0.4, 0.6) = 0.116$, $pd(0.6, 0.4, 0.6) = 0.079$, 而 $pd(0.9, 0.4, 0.6) = 0.019$ 。因而，网络拥塞程度较重的环境中，D²TCP 对期限不同数据流的调整能力基本失效，变成一个公平分配方案，这不符合基于期限的拥塞控制策略的要求。

解释 D²TCP 在网络重度拥塞时失效的原因。在 150ms 到 160ms 的时间间隔， α 从 0.4 增加到 0.6，flow₁ 和 flow₄ 的紧急因子分别是 $d_1 = 0.6$ 和 $d_4 = 0.4$ ，因而 flow₁ 比 flow₄ 更紧急。然而， $pd(0.6, d_4, d_1) = 0.079$, $pd(0.4, d_4, d_1) = 0.116$ 。如图4.1 (b) 所示，当网络拥塞程度变大时，flow₁ 的窗口和 flow₄ 的拥塞窗口大小基本相同（注意， $pd(0.9, d_4, d_1) = 0.019$ ）。因而，flow₁ 在截止期限之前无法获得足够的带宽，从而无法在截止期限之前完成。实际上，需要不同截止期限的流获得不同的带宽，从而流均在截止期限之前完成。

4.3.2 基于截止期限的速率控制策略设计原则

对于 OLDI 应用，由于查询流并发数目高，网络负载常常很大。基于此，本文主张在设计基于截止期限的网络拥塞控制策略时应遵循以下原则：

- (1) 紧急原则 (PI)：截止期限近流获得的带宽应该高于截止期限远的流获得带宽。
- (2) 差异原则 (PD)：当网络负载变大时，截止期限不同的流获得的带宽的差异应该增加。

PI 表示数据流速率应该根据流的截止期限进行区分：考虑两条大小相同的数据流（长度一致），截止期限近的流应该使用比截止期限远的流获得更多的带宽，从而流在截止期限之前完成的概率会变大。事实上，PI 在新的基于 TCP 的拥塞控制方案（如 D³ 或 D²TCP）被采纳。但是，当前基于 TCP 的拥塞控制方案，忽略了拥塞和数据流带宽差异之间的关系。

PD 表示网络负载越重，流之间带宽差异性应该越大。与网络负载较轻时相比，网络负载较重时，每条数据流获得的带宽都会减少。从而，错失截止期限的可能性会变大。因此，为了尽可能的让数据流在截止期限之前完成，期限近的流应从期限远的数据流抢夺带宽。

因此，截止期限不同的两条流获得带宽的差异性在拥塞程度大的情况下应该更大。通过此方式，截止期限近的流和截止期限远的流都能在截止时间之前完成。事实上，当前基于截止期限的拥塞控制策略忽略网络拥塞和期限的关系。在形式

上，假设 $r(\alpha, d)$ 表示在网络拥塞情形为 α ($0 < \alpha < 1$)，紧急因子为 d ($d > 0$) 的情形下流的带宽。 d 越大表示期限因子越大， α 越大表示网络越拥塞。

定理4.1是对 PI 和 PD 的数学描述。

引理 4.1： r 遵守 PI 和 PD 的充分必要条件是 $\frac{\partial r}{\partial d} > 0$ ，并且 $\frac{\partial^2 r}{\partial d \partial \alpha} \geq 0$ (但是不能恒为 0)

证明 引理4.1中第一个式子表示 PI： r 是 d 的增函数。说明期限因子越大，惩罚函数也越大，当出现拥塞时，滑动窗口减小的少。下面证明说明第二个式子表示 PD：定义 $dr(\alpha, d_1, d_2) = r(\alpha, d_2) - r(\alpha, d_1)$ 。存在：

$$\begin{aligned}\frac{\partial dr(\alpha, d_1, d_2)}{\partial \alpha} &= \frac{\partial [r(\alpha, d_2) - r(\alpha, d_1)]}{\partial \alpha} \\ &= \frac{\partial r(\alpha, d_2)}{\partial \alpha} - \frac{\partial r(\alpha, d_1)}{\partial \alpha} \\ &= \int_{d_1}^{d_2} \frac{\partial^2 r}{\partial \alpha \partial d} dd.\end{aligned}\quad (4-1)$$

考虑 PD 的数学意义，PD 等价于： $dr(\alpha_2, d_1, d_2) > dr(\alpha_1, d_1, d_2)$ ， $\alpha_1 < \alpha_2$ 并且 $d_1 < d_2$ 时成立，根据 (4-1)，当且仅当 $\frac{\partial dr(\alpha, d_1, d_2)}{\partial \alpha} > 0$ ， $d_1 < d_2$ 成立，如果有 $\frac{\partial^2 r}{\partial \alpha \partial d} \geq 0$ (但是不能恒为 0) \square

上面提出的原则简单并且有通用型，而在使用中，如何设置流的截止期限，以及如何设计由 α 和 d 作为参数的惩罚函数，可以根据具体的应用来确定。在下一小节中，本文将根据这个原则开发一个具体的根据截止期限和网络拥塞函数进行拥塞控制的策略 – 正比负载差分策略。

4.4 正比负载差分策略 (LPD) 以及分析

4.4.1 正比负载差分策略 (LPD)

作为“越拥塞，越区分”原则的一个应用，我们提出了下面的基于期限的拥塞控制算法：

$$w = \begin{cases} w + (1 - f) & \text{没有拥塞} \\ w \times (1 - f) & \text{出现拥塞} \end{cases} \quad (4-2)$$

其中 $f = \alpha/d$ 是惩罚函数，这个函数是根据网络负载 α 和紧急因子 d 进行计算的，这个因子可以用来区分流的紧急程度。拥塞程度（即网络拥塞负载因子 α ）的

含义和 DCTCP 以及 D²TCP 中的相同^[16,23]。由于 f 与 α 成正比，所以我们称这个算法为正比负载差分策略（Load Proportional Differentiation，简称 LPD）。

定义一个简单的逼近因子 $d = t_{max}/t$ ，其中 t 是流开始时间时刻和截止时刻之间的持续时间， t_{max} 是 t 的可调上界，后面会有详细讨论。对于没有截至期限的流， t 可以设置为 t_{max} 。可见， t 越小， d 越大，因此流也越迫切。基于此，可以得到 LPD-t，一个根据网络拥塞和截止期限进行拥塞控制的策略：

$$w = \begin{cases} w + (1 - \alpha \times \frac{t}{t_{max}}) & \text{没有拥塞;} \\ w \times (1 - \alpha \times \frac{t}{t_{max}}) & \text{出现拥塞} \end{cases} \quad (4-3)$$

根据 (4-3) 我们可以看到 LPD-t 是一个简单策略：首先计算惩罚函数 $f = \alpha \times \frac{t}{t_{max}}$ ，根据 f 的值来调控拥塞窗口。如果网络中没有拥塞，那么拥塞窗口每个 RTT 增加 $(1 - f)$ ，如果发生拥塞，那么拥塞窗口减小 f 倍。

4.4.2 LPD 分析

4.4.2.1 LPD 简单分析

在类似 TCP 的方案中，通过调整拥塞窗口来间接的调整流的发送速率。因此，我们可以通过调整拥塞窗口的大小来调整发送速率。LPD-t 是符合我们提出“越拥塞，越区分”的设计原则的，引理4.2对此进行了证明。

引理 4.2： LPD 服从 PI 和 PD.

证明 $\frac{\partial(1-f)}{\partial d} = \frac{\partial(1-\alpha/d)}{\partial d} = \frac{\alpha}{d^2} > 0$ ，并且 $\frac{\partial^2(1-f)}{\partial d \partial \alpha} = \frac{1}{d^2} > 0$ ，因此 LPD-t 窗口的限制，满足引理 4.1，因此 LPD-t 满足 PI 和 PD。 \square

从几何角度来看 LPD 遵循 PI 和 PD 这两个原则。由于 f 与 α 成正比，所以 (α, f) 的几何表示是一条穿过原点的直线，窗口变化服从线性规则。如果两条流有不同的期限因子 d ，那么两条数据流的惩罚函数 f 之间的差会随着 α 的增大而增加。引理4.3展示在两条流的场景下，LPD-t 稳定速率与期限成反比。

引理 4.3： 假设 LPD-t 可以使流收敛在一个稳定的状态，即拥塞窗口在一个固定的区间内稳定的波动，此时流有稳定的发送速率，并且计算出的拥塞程度趋于稳定状态。如果 $t \ll t_{max}$ ，其中 t 和 t_{max} 是在 (4-3) 中 LPD-t 使用的调整 t_{max} 参数。那么近似地，流达到的稳定速率与之对应的期限成反比。

证明 在稳定状态中，流的拥塞窗口变化规律是固定周期为 N 的锯齿形。使用 W_{min} 和 W_{max} 来代表一个周期内的最小和最大窗口，我们有：

$W_{max} = W_{min} + N \times (1 - \alpha \times \frac{t}{t_{max}})$, 并且 $W_{min} = W_{max} \times (1 - \alpha \times \frac{t}{t_{max}})$, 根据上面，我们有 $W_{max} = N \times (1 - \alpha \times \frac{t}{t_{max}}) / (\alpha \times \frac{t}{t_{max}})$, 并且平均窗口大小为： $W_{av} = (1 - \alpha \times \frac{t}{2t_{max}}) \times W_{max}$ 。

对于两条流， f_1 和 f_2 。两条流的持续时间为 t_1 和 t_2 , 假设两条流的最大窗口是 W_1 和 W_2 , 流稳定的发送速率为 r_1 和 r_2 , 两条流的速率比例， $\frac{r_1}{r_2}$ 可以计算如下：

$$\begin{aligned}\frac{r_1}{r_2} &= \frac{(1 - \alpha \times \frac{t_1}{2t_{max}}) \times W_1}{(1 - \alpha \times \frac{t_2}{2t_{max}}) \times W_2} \\ &= \frac{t_2}{t_1} \times \frac{2t_{max} - \alpha \times t_1}{2t_{max} - \alpha \times t_2} \times \frac{t_{max} - \alpha \times t_1}{t_{max} - \alpha \times t_2}\end{aligned}$$

如果 $t \ll t_{max}$, 然后 $r_1/r_2 \approx t_2/t_1$. □

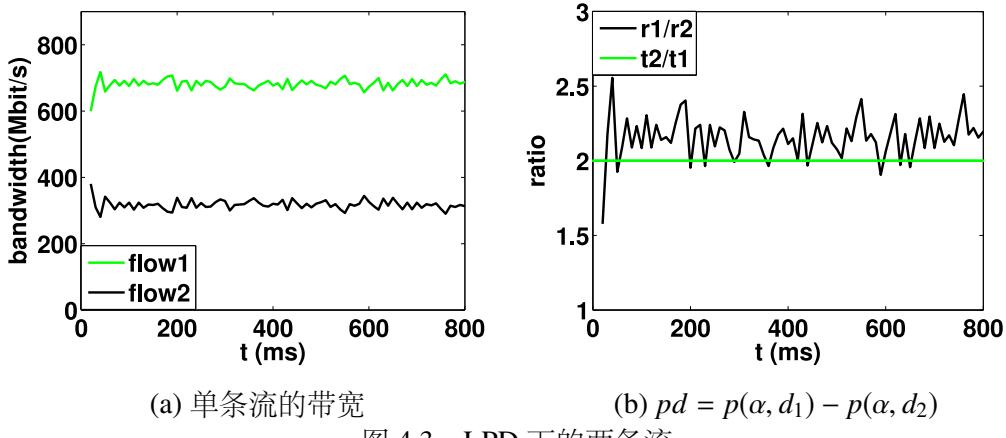


图 4.3 LPD 下的两条流

尽管 LPD-t 的稳定性还没有得到证实。实际上，流截止期限不能比 t_{max} 小得多，为了验证引理4.3，进行以下验证：启动两条均为 100 MB 的 LPD-t 长流，两条流的截止时间为 $t_1 = 5s$, $t_2 = 10s$, 设置 $t_{max} = 50s$ 。图4.4 (a) 描述了前 800 ms 流的带宽变化过程，可以看到， $flow_1$ 的带宽平均是 670M/s，而 $flow_2$ 的带宽平均是 330M/s。图4.4 (b) 描述了两条数据流实际带宽比率，可以发现 $r_1/r_2 \approx t_2/t_1=2$ 。

4.4.2.2 LPD 流模型

假设 N 个并发的流通过容量为 C pkts / sec 的瓶颈链路传递给同一个接收者节点。设置 (3-1) 中 $f_1(\alpha) = f_2(\alpha) = \alpha \times \frac{t}{t_{max}}$, 同时把 $f_1(\alpha)$ 和 $f_2(\alpha)$ 带入 (3-2)~(3-4),

并且改动(3-5)中数据包标记的方法，并且增加 α 标记方法，那么LPD流模型可以描述为：

$$\frac{dw_i}{dt} = \frac{1 - \alpha_i(t) \frac{t_i}{t_{max}}}{R(t)} - \frac{w_i(t) \alpha_i(t) \frac{t_i}{t_{max}}}{R(t)} p(t - R^*) \quad (4-4)$$

$$\frac{d\alpha_i}{dt} = \frac{g}{R(t)} (p(t - R^*) - \alpha_i(t)) \quad (4-5)$$

$$\frac{dq}{dt} = \sum_{i=1}^N \frac{w_i(t)}{R(t)} - C \quad (4-6)$$

$$\hat{p}(t) = 1_{\hat{q}(t)>1} \quad (4-7)$$

其中， w_i 表示 $flow_i$ 的拥塞窗口的大小。 t_i 是 $flow_i$ 的截止期限， α_i 表示 $flow_i$ 的拥塞程度。 q 是交换队列长度， R 表示RTT。公式(4-5)和(4-7)描述了交换机的ECN标记过程。(4-4)模拟的是拥塞窗口。公式(4-6)描述交换机队列长度。利用该模型，可以推算出窗口大小和队列长度。

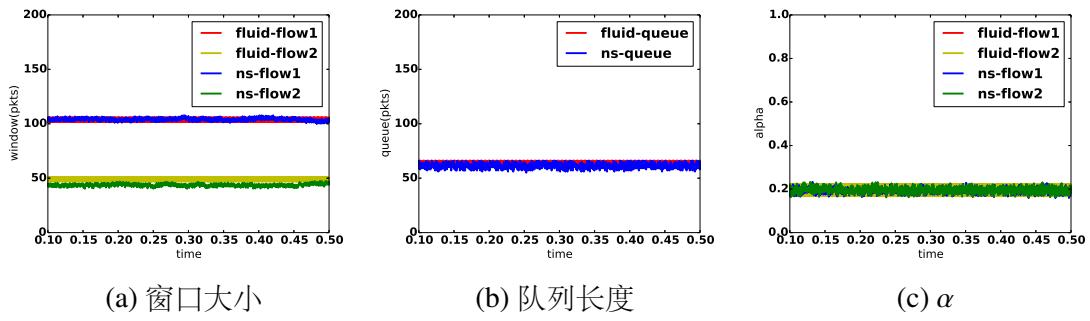


图4.4 LPD流模型和ns-2下结果对比

为了验证模型，启动两条长流，期限分别是 $t_1 = 10s$, $t_2 = 20s$ ，同时设置 $t_{max} = 40s$ 。使用相同的参数在ns-2和流体模型下对比窗口大小，队列长度和拥塞程度。窗口大小，队列长度和拥塞程度的仿真结果与模拟结果对比分别如图4.4 (a)，图4.4 (b) 和图4.4 (c) 所示。可以看到使用流模型的推算结果和ns-2的模拟结果基本相符。

为了进一步测试模型的准确性，启动两条长流，设置瓶颈带宽是10Gbps，两条流的期限分别是 $t_1 = 0.5 s$, $t_2 = 1s$ 。设置数据包大小为1500B，RTT为100 us，K为64。图4.5显示了模拟结果和ns-2方针结果的对比。从图4.5 (a) 可以看出，当 $t_{max} < 100$ 时，流模型和ns-2仿真的结果都是 $w_1/w_2 \approx 2$ 。然而，当 $t_{max} > 100$ 时，ns-2模拟的 $w_1 \approx w_2$ 和流模型的 $w_1/w_2 \approx 2$ 。图4.5 (b) 表明当 $t_{max} > 100$ 时，流

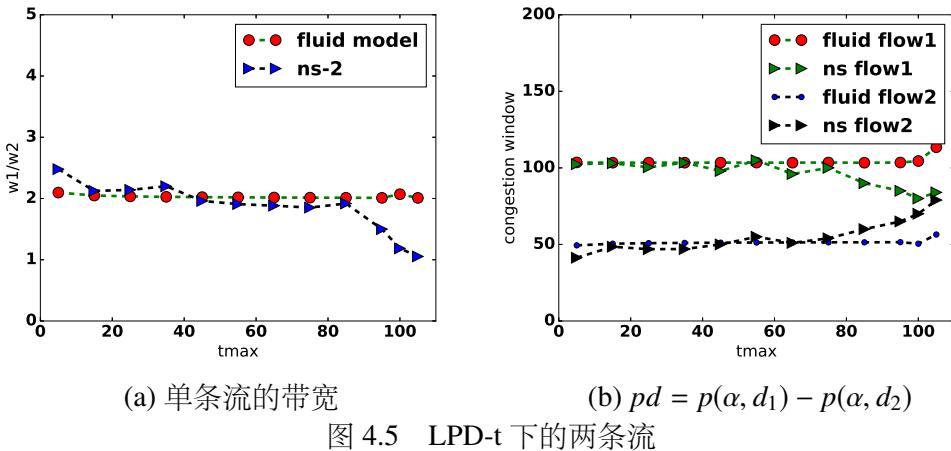


图 4.5 LPD-t 下的两条流

模型会推算出较大的拥塞窗口。此表明， t_{max} 在实际中应该被设置适当值，如何 t_{max} ，本章后续会进行更详细的讨论。

4.4.2.3 t_{max} 的选择

注意，在 LPD 流模型中，流具有不动点 $(w_i, \alpha_i) = (t_{max} - t_i, 1)$ 。不动点的实际意义是所有数据流的 $\alpha = 1$ 。即重度拥塞的情形下，所有的数据包都被标记，所有流拥有固定的拥塞窗口。在 LPD 中， t_{max} 应该选择适当的值。下面分析 t_{max} 取值：

假设有 N 条流 ($t_{max} > t_1 > t_2 > \dots > t_n$) 同时启动，其中， $t_1 \sim t_n$ 是流 $flow_1 \sim flow_n$ 的截止期限。在时刻 t，队列长度表示为 $q(t)$ ，则有：

$$\frac{t_{max} - t_1}{t_1} + \frac{t_{max} - t_2}{t_2} + \dots + \frac{t_{max} - t_n}{t_n} < q(t) + C \times RTT \quad (4-8)$$

假设当交换机队列超过 K 时，数据包开始标记。那么，对于 N 条数据流，每个 RTT，拥塞窗口大小不超过 1。那么我们有 $q(t) < K + N$ 。因为我们假设 $t_{max} > t_1 > t_2 > \dots > t_n$ ，因此有：

$$t_{max} < \frac{t_1 \times (K + C \times RTT + 2N)}{N} \quad (4-9)$$

根据 (4-9)，我们可以根据截止期限的最大值，最大并发流量数量，RTT 和瓶颈链路链接带宽来设置 t_{max} 。在当前数据中心，设置 C 为 10Gbps，数据包大小为 1500B，RTT 为 100us，因为数据中心，大多数情况下并发连接数小于 100^[16]，假设设置 K 为 60：

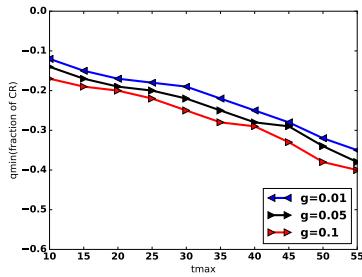


图 4.6 队列最小长度波动

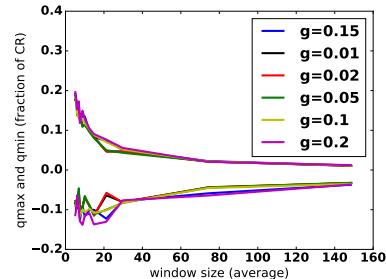


图 4.7 队列最大和最小

$$t_1 < t_{max} < 3.5 \times t_1 \quad (4-10)$$

在实际部署 LPD 时，可以根据 (4-9) 计算 t_{max} 。 (4-10) 可以当作 t_{max} 的缺省值。事实上，结果与图4.5 (b) 一致。根据 (4-10)，当 $N = 2$ 时， $t_{max} \leq 76 * t_1$ 。在此情况下， $t_1 = 1s$ ，从 ns-2 仿真中可以看出，当 $t_{max} > 80$ 时，流拥塞窗口不再与其对应的 t 成反比。

4.4.2.4 K 的设置

实际中，交换机的阈值 K 不可设置的过大或者过小，否则，链路利用率会很低。在实际部署 LPD 时，需要使链路利用率达到 100%。因此需要确定阈值 K 的设置范围^[52]。

q_{min} , q_{max} 表示交换机队列的最小和最大长度。基于 ECN 标记的拥塞控制策略，队列长度在固定区间内波动。阈值 K 应大于下冲振荡（否则队列大小将为零，链路利用率将不是 100%）。使用 LPD 流模型，并将流的期限设置在 10 以内，将 t_{max} 从 10 变为 55，并将链路带宽从 1Gbps 变为 10Gbps。最后，针对每一个 t_{max} 我们记下每组队列下限的最小值。结果如图4.6所示。可以看到 g 值很小，即使是最坏的情形 ($t_{max} = 55, t_{max} \approx 5t_1$)，也可以得到：

$$K \gtrsim 0.4CR \quad (4-11)$$

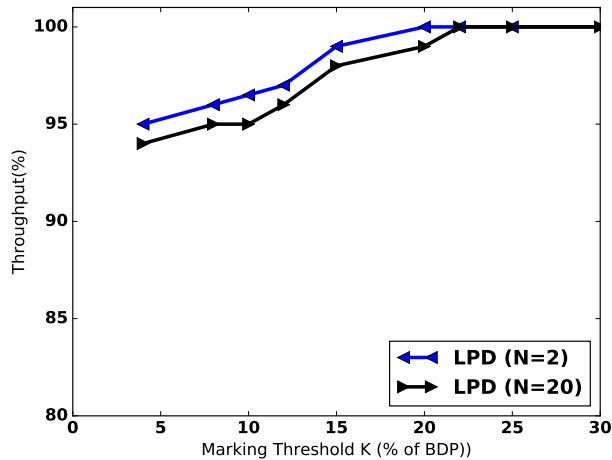


图 4.8 ns-2 仿真中吞吐量和标记阈值 K 的关系。从 4 到 30 个包（BDP 的 2.4 – 18%）变化。结果表明，即使对于小于 BDP 的 3% 的 K，吞吐量仍然高于 95%

至于 RTT，因为 $R = d + K/C$ ，将 R 带入 (4-11)，所以在最坏的场景下，得到队列的缺省值 $0.67Cd$ 。假设链路带宽是 $C = 10\text{Gbps}$ ，链路传送延迟 $d = 100\mu\text{s}$ ，数据包大小是 1500B ，则 K 至少应该设置为 56。对于数据中心中常见的情形，链路传播延迟为 $100\mu\text{s}$ ，链路的瓶颈带宽为 10Gbps 。设置 $t_{max} = 10$ ，截止期限在 1 到 10 范围内随机生成。使用 LPD 流模型，图4.7显示了 flow_1 ，使用不同的 g 值下平均窗口大小与 q_{min} 和 q_{max} （其中 q_{min} 表示队列最小值， q_{max} 表示队列最大值， q_{min} 和 q_{max} 换算成 CR）的关系图。可以看到 $K \gtrsim 0.12CR$ 。为了避免队列下溢，需要有 $K > |q_{min}|C * R \approx 0.12CR$ 。由于 $R = d + K/C$ ，因此可以得到：

$$K \approx 0.14Cd \quad (4-12)$$

换言之，大约需要 14% 的 BDP 就能实现链路利用率达到 100%。多余的缓冲区可以用来处理突发流。为了测试上面的实验结果，图4.8显示 ns-2 仿真中吞吐量和标记阈值 K 的关系图。在这个实验中，变换 K 的取值从 4 到 30（大约是 BDP 的 2.4% ~ 18%）。结果表明，ns-2 仿真结果与 LPD 流模型相似，即使当 K 设置小于 3% 倍的 BDP 时，吞吐量仍然高于 95%。

4.4.3 LPD 拓展

对期限敏感的 OLDI 应用常为小的查询流设置期限。为了满足 OLDI 应用中有限期数据流的需求，我们扩展 LPD-t，并将流的大小考虑进去。

$$f = \alpha \times \frac{t}{t_{max}} \times \frac{s}{s_{max}} \quad (4-13)$$

其中， s_{max} 是流大小的上界。这样，在链路带宽不足的场景下，小的数据流能获取更多的带宽，从而在截止时间（deadline）之前完成。(4-13) 仍然保留了 LPD 的特性，称此为 LPD-e。在本文后面的实验中，默认使用 LPD-e，如果本文使用其他的策略，将会有明确说明。可以注意到“越拥塞，越区分”的原则是通用的，因此可以使用其他的形式。本文选择仅仅因简单性而提出 LPD，并将在后面中讨论一些其他形式的拥塞控制策略，其它形式的策略也满足此原则。另一方面，LPD 本身也可以使用剩余流大小进行拓展。

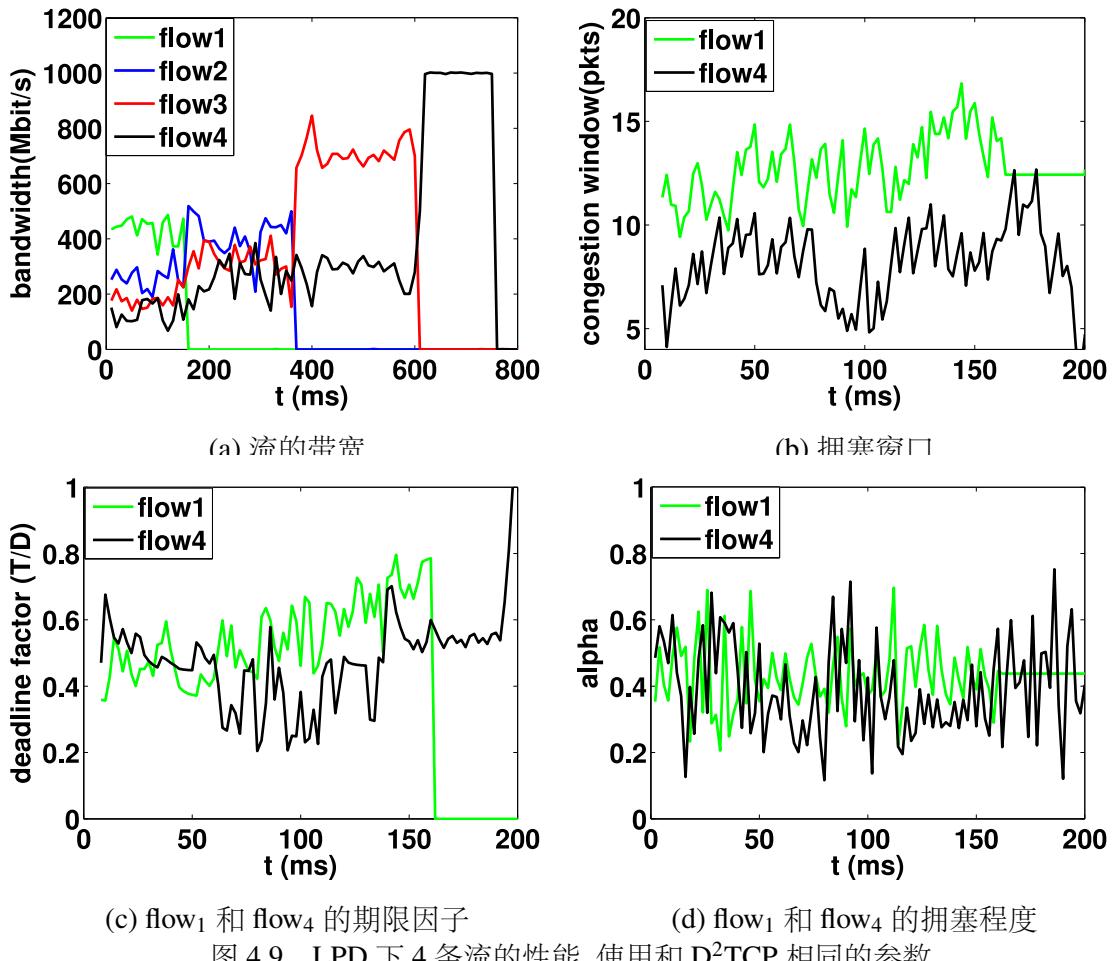
4.5 实验验证

本节通过 ns-2^[48] 仿真和实际测试床来评估 LPD 的性能。本文根据 DCTCP 网站^[53]发布的 DCTCP 的源代码实现了 LPD（包括 LPD-t, LPD-e），并将 Vamanan^[23]提供的 D²TCP 的 ns-3 代码移植到 ns-2。另外，本文的实验同时实现 L²DCT, Karuna，并改动 pFabric 的代码到 ns-2 平台上。本文所有代码均可以在 LPD 开源代码网站^[49]下载。同时本文的实验也在 Linux 内核 3.2.61 中实现了 LPD, D²TCP, L²DCT，并在真实环境上作评估，相应的源代码可以在 LPD 内核代码网站^[50]下载。

4.5.1 解决研究动机提出的问题

本文首先将 LPD 应用到4.2.2节研究动机介绍的实例中。虽然此例不是一个典型的 OLDI 场景，但是可以用它来检查 LPD 是否能够帮助更多的流量达到最终期限，特别是在负载较重的情况下测试 LPD 的性能。使用与 D²TCP 相同的期限因子 $d=T/D$ ，改动 LPD，称之为 LPD-d。在相同的设置下，如图4.9 (a) 所示，只有 flow₂ 在 LPD-d 下错失期限（deadline），为了更好地理解 LPD 优于 D²TCP 和 DCTCP 的原因，我们在图4.9中给出了 LPD 下流状态的更多细节。

图4.9显示每条流的带宽，以及拥塞窗口 w ，期限因子 d 和 flow₁ 与 flow₄ 的网络拥塞程度 α 的情形。与图4.1对比可以看出，随着流发送，flow₁ 和 flow₄ 之间拥塞窗口之间的差异变得更大，而 flow₁ 和 flow₄ 的期限因子和网络负载变化不大。这是因为选择正比负载差分策略，当网络拥塞程度增加时，flow₁ 和 flow₄ 有更大的

图 4.9 LPD 下 4 条流的性能, 使用和 D²TCP 相同的参数

带宽分配差异。如图4.9 (a) 所示, 最终, LPD-d 比 D²TCP 错过期限的流的数目更少。

为了进行比较, 设置 $t_{max} = 800\text{ms}$, 并且在相同的实验环境中测试 LPD-t。如图4.10 (a) 所示, 使用 LPD-t, 所有流在截止期限之前完成传输, 这是因为使用 LPD-t, 截止时间更近的流会获得更多的带宽, 例如有带宽大小关系: flow₁ > flow₂ > flow₃ > flow₄。图4.10描述了 LPD-t 内部状态, 与 LPD-d 和 D²TCP 相比, 如图4.10 (c) 与图4.10 (d) 所示, 尽管流计算出的网络负载 (即每条流计算的 α 值) 基本相同, 但如图4.10 (d) 所示, 不同截止期限的数据流的拥塞窗口差异很大。例如, 在前 200 ms 时间内, flow₁ 获得约 55% 的带宽, 而 flow₄ 仅获得 10% 的带宽。图4.10 (c) 显示在开始 200ms 时间内交换机的队列长度, 可以看到, 交换机的队列在阈值 $K = 25$ 附近表现出小的波动。队列在 128ms 处突然下降是由于 flow₁ 在此时间点完成传输, 此后, 队列很快被未传输完成的数据包填充, 由此还可以看到 LPD 数据流可以很快的填充交换机队列, 从而充分利用链路带宽。

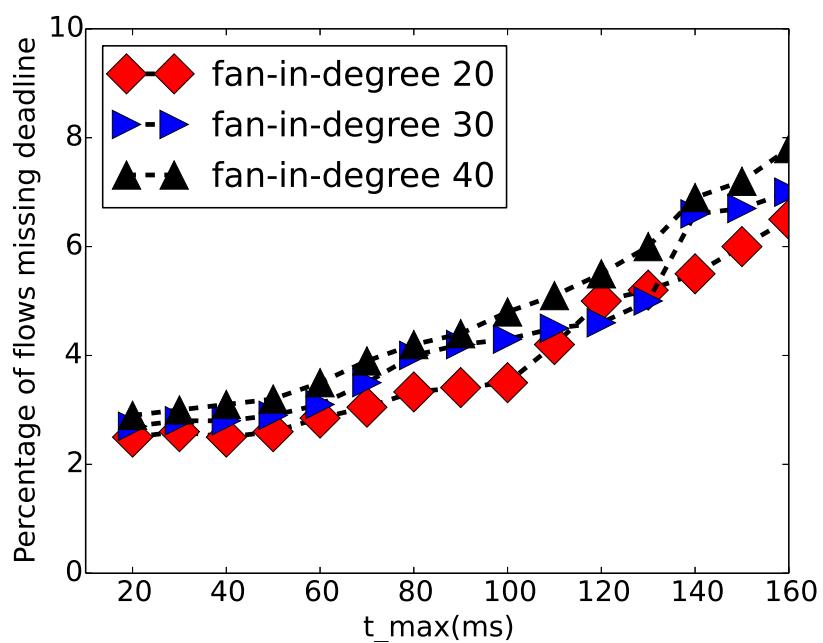
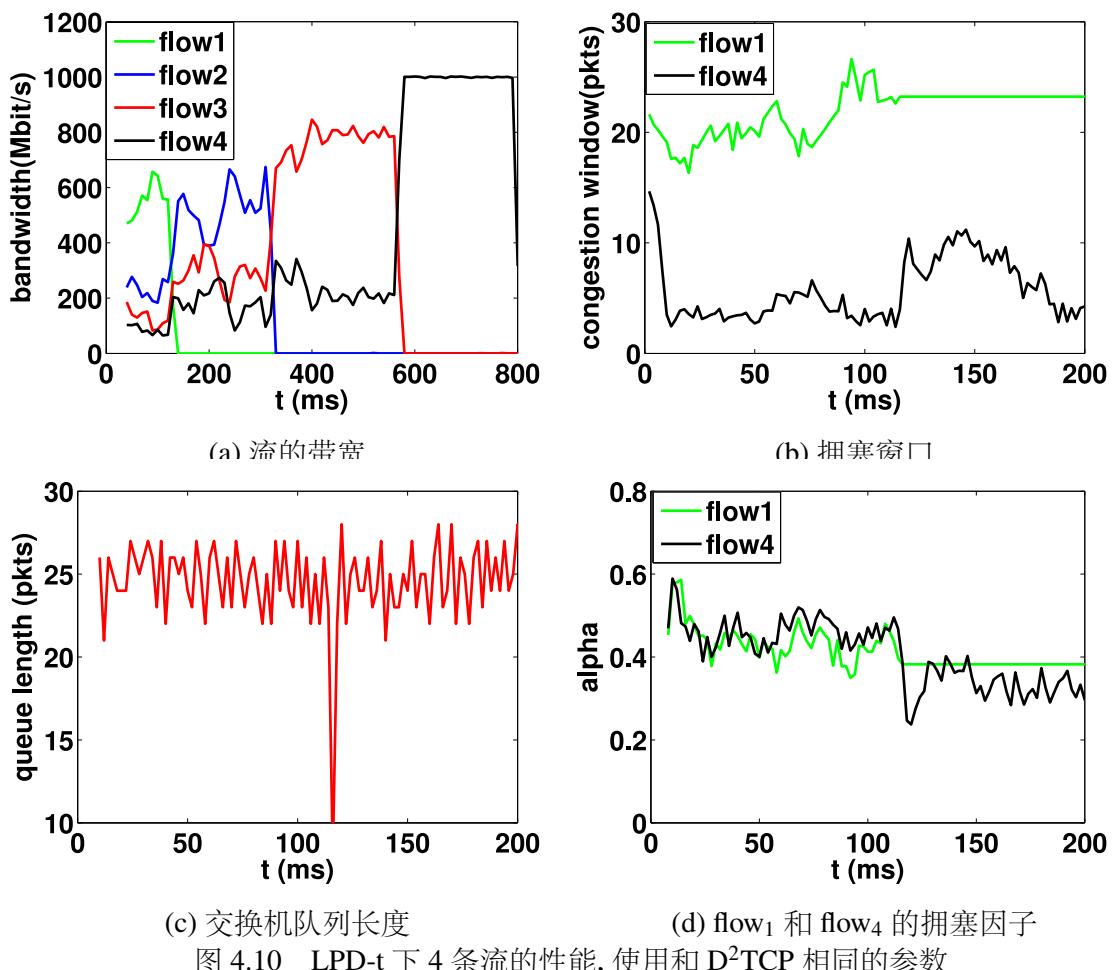


图 4.11 变化 t_{max} 的影响

4.5.2 参数选择

LPD-t 和 LPD-e 必须设置适当的上限 t_{max} 和 s_{max} 。此处本文只讨论 t_{max} ，因为 s_{max} 有类似的作用。从 LPD-t 的等式 (4-3) 可以看出如果 t_{max} 变大，惩罚函数 f 就变小。从 (4-9)，得知对于 $C = 1\text{Gbps}$, $K = 25$, $\text{RTT} = 1\text{ms}$, 最大数据包大小为 1500B ，当并发数据流数量为 $20, 30, 40$ 时，对应的 t_{max} 应小于 $6*t_1, 4*t_1, 3*t_1$ 。如果 t_{max} 太大，区分不同期限的流的能力也变弱。使用 OLDI 应用来检查 t_{max} 对 LPD-t 的性能影响，其中流的截止期限都设置在 20ms 内，变换 t_{max} 从 20ms 变化到 160ms (即 $t_1 < t_{max} < 8*t_1$)。图4.11描述了不同 t_{max} 下错失期限的流数目，而扇入度 (fan-in-degree, 即同时到达根节点的流的数目) 分别是 $20, 30$ 和 40 。

从图4.11可以看出，当 $t_{max} < 80\text{ms}$ 时，即 t_{max} 不超过最大截止期限的 4 倍时，使用 LPD-t，少于 5% 的流错过截止期限。LPD-t 的性能总是比 D²TCP 优异 (D²TCP 在扇入度分别是 $20, 30$ 和 40 时错过期限的比例分别为 $3.45\%, 4.96\%$ 和 5.45%)。而且在这个范围内，使用不同 t_{max} ，LPD 的性能差异较大。然而，当 t_{max} 很大时，例如，当 $t_{max} > 120\text{ms}$ (即 6 倍于截止期限) 时，LPD-t 的性能受到极大的损害，并且比 D²TCP 差。

4.5.3 仿真测试

4.5.3.1 简单拓扑测试

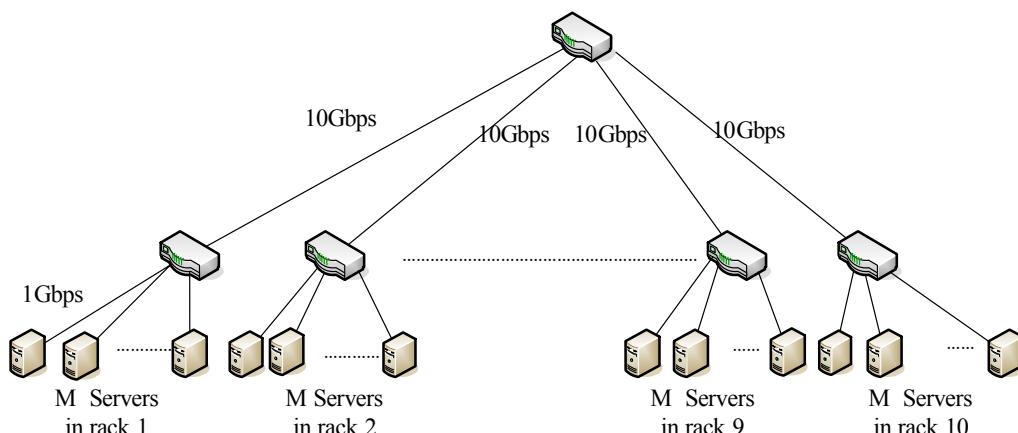


图 4.12 简单的数据中心拓扑

为了测试在数据中心下 LPD-e 的性能，如图4.12所示，我们构建了一个三级树形拓扑。在这个拓扑中，我们构建 10 个机架，每个机架由 $M = 40$ 个服务器组成，机架连接到顶层的 TOR 交换机。服务器和交换机之间的链路带宽为 1Gbps 。所有 TOR 交换机通过 10Gbps 链路连接到根交换机。在 ns-2 中构建这种三级树形拓扑，

在这个中等数据中心拓扑下，我们评估 LPD-e 的性能。

和之前工作^[16,23,24]类似，设置交换机的标记阈值 $K=20$ ，并将传播延迟 d 设置为 100us。TCP 重新传输超时（RTO）设置为 10ms，每个发送端初始拥塞窗口设置为 12。考虑在数据中心内，很多大小是 200 KB 以内的数据流是有期限的，因此设置 $s_{max} = 200KB$ 。假设流的大小遵循 Pareto 分布，Pareto 参数为 1.2，流的平均大小为 50KB。将流截止期限设置为 20, 30 和 40 ms。对于 LPD，将 t_{max} 设置为 80 ms。假设 OLDI 应用的扇入度从 20 到 40 不等，OLDI 的扇入度代表了不同程度的网络拥塞。此外，在所有模拟的服务器中，使用 10 台服务器产生背景流量。由于查询结果被返送给 OLDI 应用程序的根节点，而且所有有期限的流的启动时间基本相同。所以，如果使用传统的 TCP，因为大量并发的小流量共享一个瓶颈链路往往会导致严重的拥塞，因此可能会导致 Incast 问题^[21]。

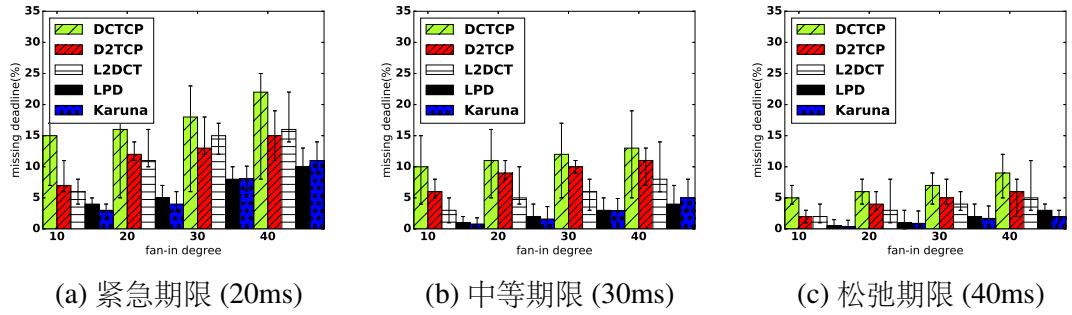


图 4.13 DCTCP, D²TCP, L²DCT ,Karuna 和 LPD-e 下 OLDI 应用错失期限的比例对比

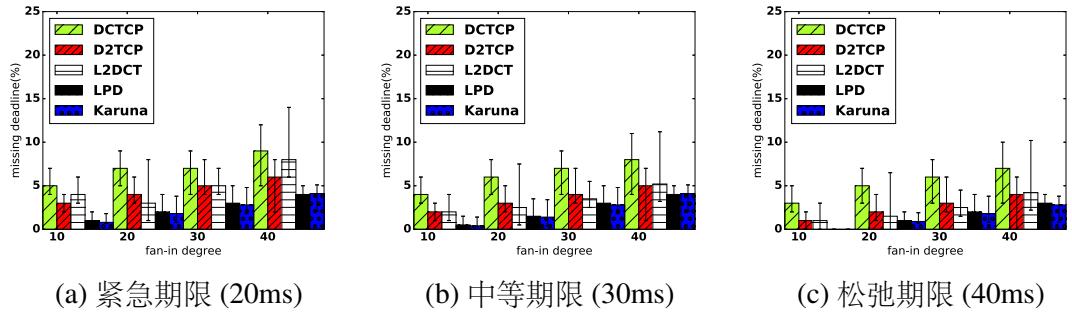
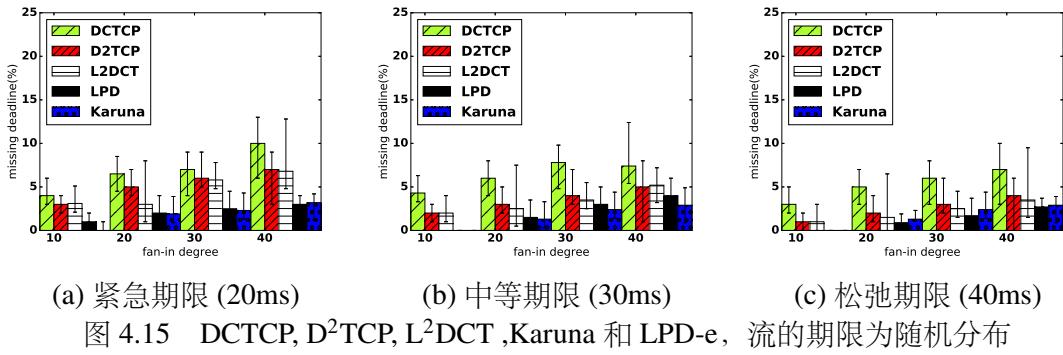


图 4.14 DCTCP, D²TCP, L²DCT ,Karuna 和 LPD-e，流大小分布为 Pareto 分布的错失期限比例

图4.13显示了不同扇入度下流错失期限的百分比。为了更好的展示 LPD 的性能，本文使用 DCTCP, D²TCP, L²DCT, Karuna 和 LPD-e 作为对比方案。变换每组实验参数 100 次，图4.13显示的结果中，每个柱状图显示错失期限流的百分比的平均值，其中 error bar 展示了错失期限的流百分比最小值和最大值。注意到，L²DCT 也是一种基于 TCP 的拥塞速率控制机制。与 D²TCP 类似，L²DCT 也使用 gamma



函数进行拥塞窗口的计算，但是和 D²TCP 不同的是，L²DCT 只使用流大小作为拥塞窗口的调整因子。实验结果表明 LPD-e 在网络拥塞程度很重的情形下依然可以根据流的期限进行区分，因此，相比 DCTCP, D²TCP, L²DCT, Karuna 等，使用 LPD-e 使得流错失期限的数目最少。尽管 D²TCP (以及 L²DCT) 在 DCTCP 上有所改进，如图4.13 (c) 所示，D²TCP 仅仅在期限较宽松的情况下性能比 DCTCP 有明显的优势，但对期限很紧的情形，如图4.13 (a) 和 (b) 所示，D²TCP 并没有明显的优势。事实上，对此问题，D²TCP^[23] 并没有进行详细讨论。对于最先进的基于截止期限进行速率控制的方法 Karuna^[34]，性能平均比 LPD-e 好 5%。然而，对于并发程度很高的情形（如扇入度为 40），此时网络处于重度拥塞，Karuna 的性能比 LPD 差。其原因是随着网络拥塞程度增加，使用 LPD，不同期限的流获得带宽差距越来越大，从而，数据流获得不同带宽，因而，截止期限更近的流能获取更多的带宽，从而更有可能在截止期限之前完成。

在另一种不同的场景下测试 LPD-e 性能，在此场景下，服务器之间相互发送短流，变化每台机架上服务器的数目，设置短流的到达间隔服从泊松过程（每台服务器每秒钟发送 1000 条数据流），而所有其他设置（包括流量大小分布和截止期限）与上述 OLDI 场景中的设置相同。图4.14中描绘每个策略流错失期限数目的百分比。图中柱状图每个柱呈现的结果包括平均期错过期限的流百分比，以及进行 100 次随机实验得到错过期限流的数目的最小和最大值，其中我们可以看到 LPD 在所有情况下，性能比其他的方法要好。特别是当负载相对较轻的情况时（即每个机架上的 10 台服务器发送短流，此时网络拥塞程度较轻），无论截止期限是近还是远，使用 LPD-e 错过期限的数目都是 0。与其他协议相比，除了当网络负载很重的时（即每个机架有 40 个服务器发送短流量时），在几乎所有情形下，使用 LPD-e，错过截止时间的流的数目降低 50% 以上。在负载很重时，与其它协议相比，LPD-e 仍然可以将错失期限的流的比例降低至少 30%。

假设流大小在 2 KB 到 200 KB 范围内均匀分布，流的期限服从平均值为 30 ms, 40ms 和 50ms 的指数分布。其它实验参数和上述其它实验相同，实验结果结

果如图4.15所示，不管拥塞的程度是轻微，中等还是重度，LPD-e 仍然性能最好。注意到，在所有的仿真结果中，L²DCT 通常可以达到与 D²TCP 相当的结果，猜测这是由于它们都使用了伽马校正功能。此外，在大多数情况下，Karuna 表现比 LPD-e 更好，但是在网络负载较重的情况下，有时 LPD-e 可以比 Karuna 表现更好，这是因为 LPD-e 的设计中遵循“越拥塞，越区分”的原则，当网络拥塞程度严重时，不同期限的数据流获得的带宽区分很大，因此更多紧急的数据流能在截止期限之前完成。

4.5.3.2 复杂拓扑下测试

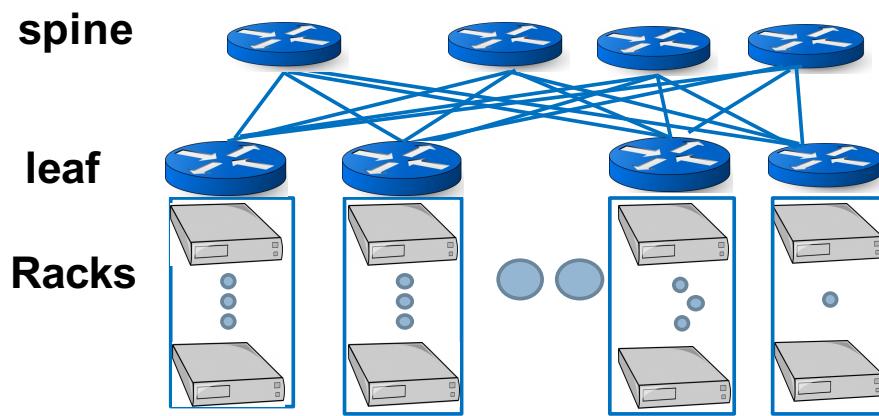


图 4.16 数据中心 spine-leaf 拓扑

为了展示 LPD 如何在复杂的数据中心拓扑结构下的性能，以及 LPD 在真实流量下的性能，我们在 ns-2 模拟器中构建了 spine-leaf 拓扑，拓扑结构如图4.16所示。在构建的拓扑中，共有 144 个服务器连接到 9 个叶子交换机，这 9 个叶子交换机连接 4 个主干交换机。通过主干交换机（4 跳）的端到端往返延迟是 14.6 us，叶交换机（2 跳）的端到端往返延迟是 13.3 us。在这些延迟中，10us 的时间是在端节点上，这模拟的是像 hadoop 这类应用的计算时间。TCP 重传超时（RTO）设置为 10 ms，初始拥塞窗口设置为 12。主干交换机和叶交换机之间的链路容量为 $40Gbps/x$ ，其中 x 为超额认购因子，叶节点和终端服务器之间的容量为 10Gbps。 $K = 64$ ，设置 s_{max} 的默认值是 1MB， $t_{max}=30ms$ 。

本文在实验中使用两个世纪数据中心应用的流量：网络搜索（web search）和数据挖掘（data mining）^[16,32]。在网络搜索流中，70% 的流小于 1 MB，超过 50% 的流在 100 KB 到 1 MB 之间。在数据挖掘流中，超过 80% 的流是小于 10KB 的小流，大于 1MB 的 4% 的大流占据超过 90% 的数据量。在本部分的仿真中，假设数据流的到达服从泊松分布。流的源和目标服务器是随机选择的，并且服从均匀分布。

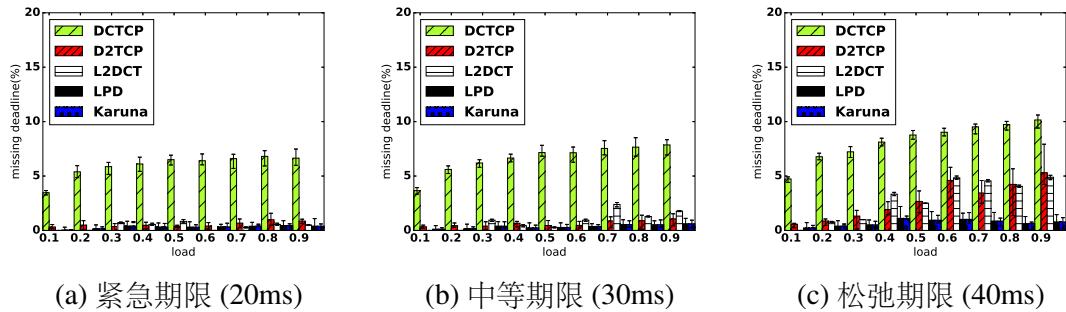


图 4.17 网络搜索应用流量下, LPD-e,D²TCP,L²DCT ,Karuna 和 DCTCP 性能对比。流的截止期限是在 10ms 到 30ms 的统一分布

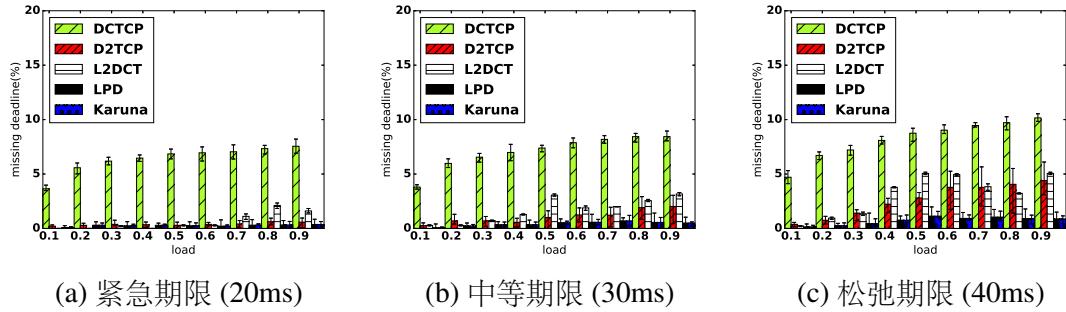


图 4.18 数据挖掘应用流量下, LPD-e,D²TCP,L²DCT ,Karuna 和 DCTCP 性能对比。流的截止期限是在 10ms 到 30ms 的统一分布

流到达的时间间隔不同, 会产生不同程度的网络负载。当今的数据中心网络往往是超额认购^[54], 在后续的实验中, 本文变更超额认购参数来模拟此。

对于截止期限比较紧的流, 本文选择错失期限的流的比例作为评价标准来评价算法。对于截止期限不敏感的数据流, 本文考虑流平均完成时间。本节测试在延迟敏感的网络负载下 LPD 的性能。在实际中, 网络搜索等对延迟敏感的业务需要在几毫秒内将计算结果传送给客户端, 否则会影响用户体验。在下面实验中, 设置截止期限在 10ms 到 30ms 之间均匀的分布。其余的参数适用默认的值。将超额认购因子 x 从 1 改变为 10, 图4.17显示了结果。需要注意的是, 每组实验重复 100 次, 在图中标记处每组的平均值, 最大值, 最小值。

从图4.17可以看出, 使用 LPD-e 有少于 1% 的数据流错失期限, 而使用 DCTCP, D²TCP, L²DCT 时, 分别有 17%, 2.5%, 2.4% 错失期限。特别是, 当负载和超额认购因子很小 ($load=0.1$, $x=1$) 时, D²TCP 和 L²DCT 错过期限的流数目是 LPD-e 的 2 \times 。但是当超额因子和负载较大 ($load=0.9$, $x=10$) 时, D²TCP 和 L²DCT 错过期限的流数目是 LPD-e 的 5 \times 。这是因为随着超额认购因子变大, 网络负载变大, LPD 的惩罚函数比 D²TCP 和 L²DCT 的 gamma 校正函数性能更好, 因此流错失期限的比例更小。Karuna, 当前最先进的基于期限的拥塞控制方法, 有不到 1% 的流错失期限。然而, 当网络的负载变重 (负载 = 0.9, $x=10$) 时, LPD 表现比卡

Karuna 好。这是因为，当网络负载加重时，使用 LPD，不同截止期限的流的拥塞窗口差异更大。使用数据挖掘应用的结果与之类似，结果如图4.18所示。

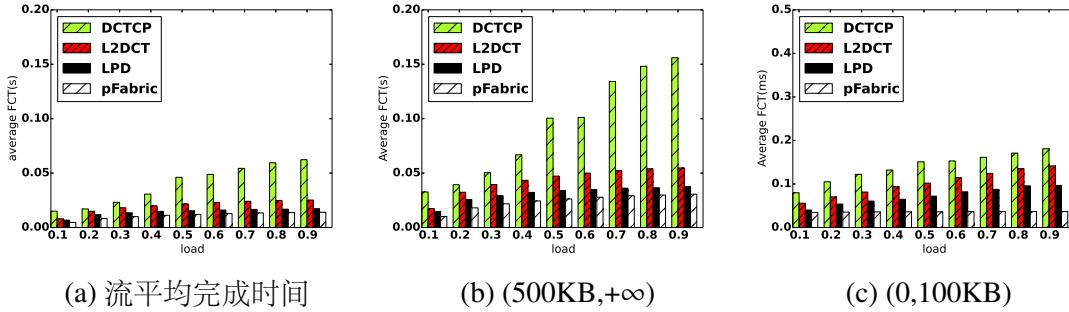


图 4.19 网络搜索流量下，LPD, L²DCT, pFabric 和 DCTCP 对流平均完成时间的对比，注意意图 (c) 和图 (a),(b) 的 Y 轴的单位不同

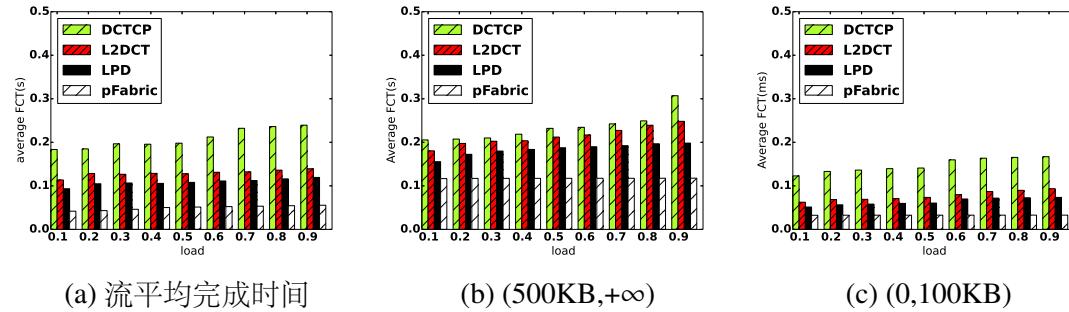


图 4.20 数据挖掘流量下，LPD, L²DCT, pFabric 和 DCTCP 对流平均完成时间的对比，注意意图 (c) 和图 (a),(b) 的 Y 轴的单位不同

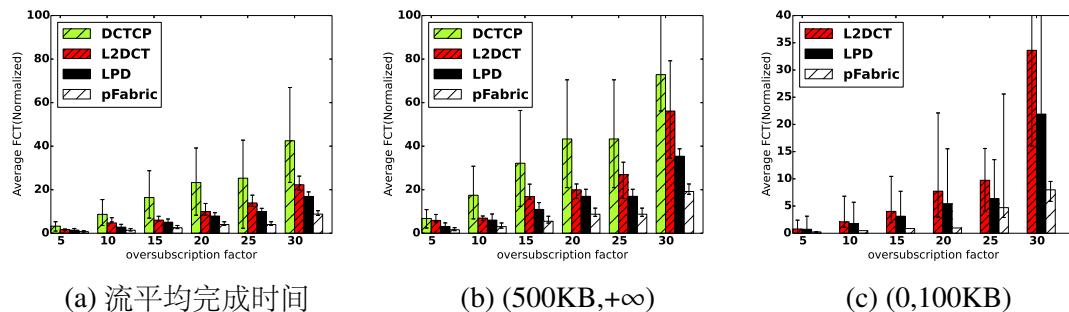


图 4.21 网络搜索流量下，LPD, L²DCT, pFabric 和 DCTCP 对流平均完成时间的对比，变幻超额认购参数从 5 到 30

“越拥塞，越区分”的原则也可以用于其他目标的优化，比如流平均完成时间（Average Flow Completion Time，简称 AFCT）。改动 LPD，并且设置惩罚函数为 $f = s/s_{max}$ ，我们可以得到优化流平均完成时间的策略 LPD-s。在 LPD-s 中， s 可以是流大小或剩余流大小。在后续实验中，默认 s 为流大小，并设置 $s_{max} = 1MB$ 。

对于大于 1 MB 的流，设置其 $s = 1\text{MB}$ 。对于小于 10KB 的流，设置 $s = 10\text{KB}$ 。假设 $N = 1$ ，根据 (4-9)，LPD-s 的流大小调整区域是 [10KB, 1MB]。设置超额订购因子 $x = 5$ ，图4.19显示了网络搜索应用下的流平均完成时间结果。

从图4.19可以看出，LPD-s 和 L²DCT 均比 DCTCP 的平均流完成时间都小，但性能不如 pFabric。图4.19 (a) 表明，对于网络负载较轻的场景 ($\text{load} < 0.5$)，LPD-s 的性能是 DCTCP 的 $3\times$ ，大约是 L²DCT 的 10%。但当网络拥塞很重 ($\text{load} > 0.5$) 时，LPD-s 性能分别比 DCTCP 和 L²DCT 提高 $5\times$ 和 15% 。图4.19 (b) 显示的是流大小大于 500 KB 的实验结果。图4.19 (c) 是小流的结果，注意到图4.19 (c) 与图4.19 (b) 和 (a) 的 y 轴有不同的单位。

图4.20显示的是数据挖掘流量下流平均完成时间对比。特别地，从图4.20 (b) 可以看出，LPD 下流平均完成时间减小约 40%，而图4.19 (b) 显示了对于长流，LPD 在流平均完成时间方面性能提高 $1\times$ 。对于数据挖掘应用，因为大部分数据流长度大于 1MB，因此在实验中，设置 $s_{max}=1\text{MB}$ ，大于 1MB 的流被认为有最低优先级，因而长流的优先级低。

为了测试 LPD 在网络负载很重的情况下的性能，我们将超额认购因子从 5 改变到 30。对于每组实验，将网络负载从 0.1 改变到 0.9，图4.21显示了平均流完成时间和流完成时间的最大值以及每组的最小值。可以观察到超额认购因子比较小 ($x=5$) 时，LPD 的平均流完成时间与 L²DCT 相比下降了 20%。但是，当网络的超额认购因子较大时，LPD 的性能比 L²DCT 提高 $1\times$ 。这个结果表明在网络拥塞重的情形下 LPD 是一个比较好的策略。

4.5.4 真实环境下测试

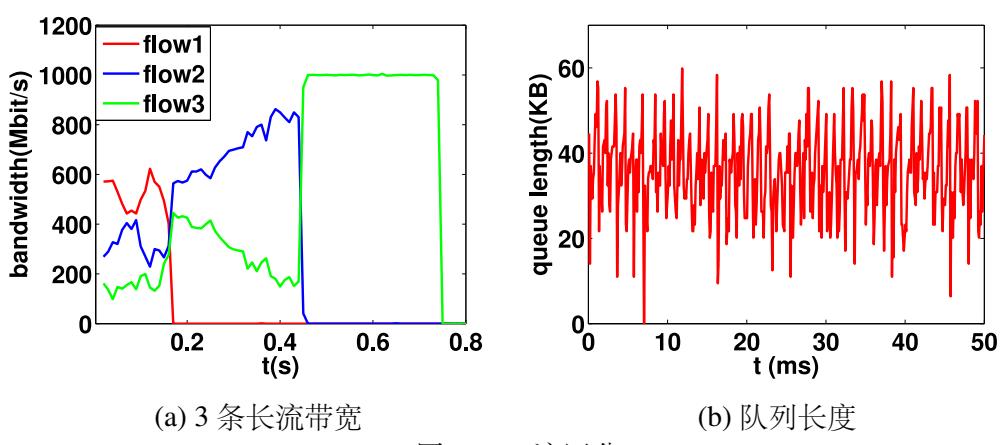


图 4.22 流区分

4.5.4.1 简单拓扑下的测试

为了在真实环境下测试 LPD 的性能，在 Linux 内核 3.2.61 中基于 DCTCP 的代码实现了 D²TCP 和 LPD-e。交换机使用 NetFPGA 来实现，并且每个交换机都有 4 个 1Gbps 以太网端口，因此数据包可以线速转发和标记。由于硬件有限，我们只能建立两台交换机和六台服务器的小数据中心拓扑结构。LPD 对 Linux 内核 3.2.61 评估的主要代码可以在^[50] 下载。

本文首先验证 LPD-e 是否可以在真实环境中区分不同期限的流量，流大小设置为 10, 30 和 50 MB，相应的截止时间为 300, 600 和 900 ms。图4.22绘制了每个流的带宽随时间的变化情形，以及交换机上的队列长度。与仿真结果类似，不同截止期限的流程明显不同，队列长度保持在 37.5 KB 的标记阈值（约为 $K = 25$ 个 1.5 KB 的数据包）。

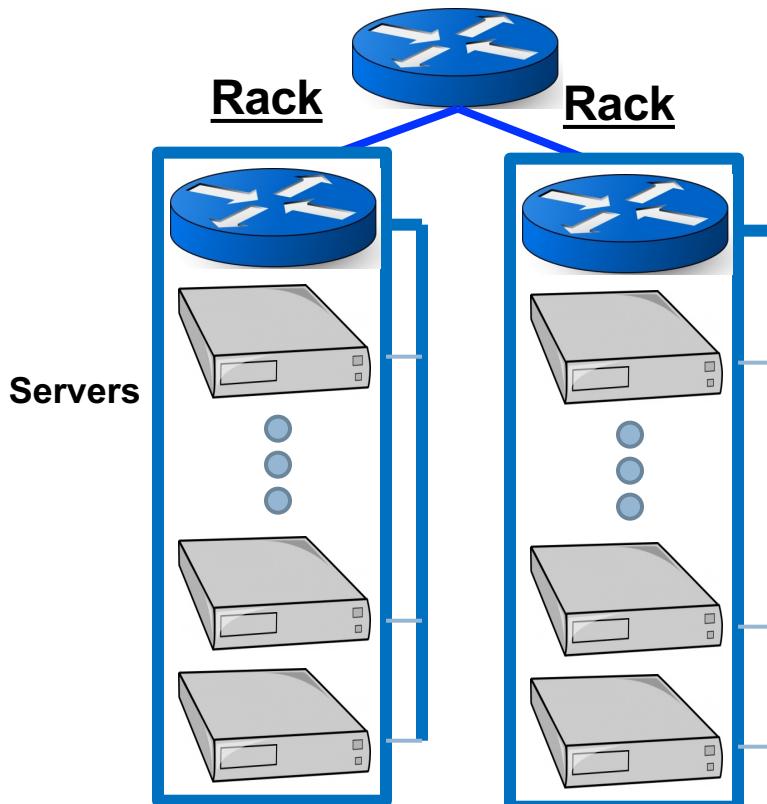
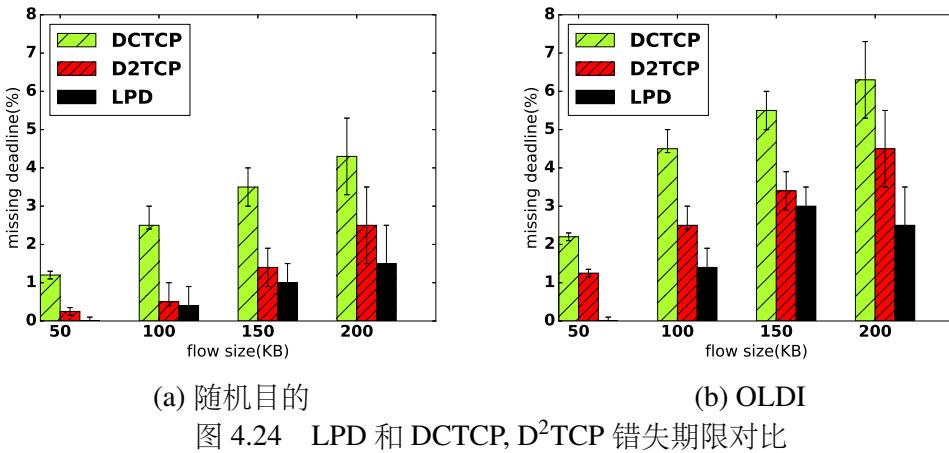
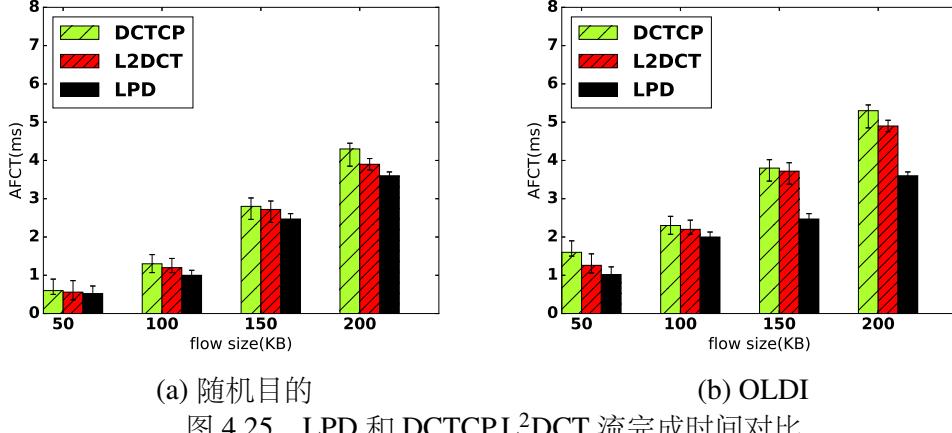


图 4.23 小型数据中心测试床

4.5.4.2 小型数据中心测试

建立一个小型的基于树的测试环境来测试 LPD 的性能。图4.23显示了拓扑结构。在此拓扑结构下，两台 Linux 服务器运行 Ubuntu14.04 以及 dummynet^[55,56] 来

图 4.24 LPD 和 DCTCP, D²TCP 错失期限对比图 4.25 LPD 和 DCTCP,L²DCT 流完成时间对比

构建一个双向瓶颈链路。修补 dummynet 用来标记数据包。在实验中，拓扑中所有接口都是 1Gbps。

LPD 的惩罚函数遵循“越拥塞，越区分”的原则，即使在网络重度拥塞的条件下也能区分期限不同的数据流。因此更多的流能够在最后截止期限前完成。在实验中，每个服务器 S_i 随机与服务器 $S_j (i \neq j)$ 建立连接。随后 S_i 连续发送数据流给 S_j 。然后记录数据流错失期限的平均值，最大值和最小值。图4.24(a) 显示的结果。可以看到 LPD 比 D²TCP 和 DCTCP 更好。与 DCTCP 和 D²TCP 相比，LPD 下流错失期限的比例减少了约 50% 和 30%。为了观察在 OLDI 应用下 LPD 的性能，让 $S_1 \sim S_6$ 连续向根 R 节点发送随机截止期限 ($\leq 10\text{ms}$) 的小流，图4.24 (b) 显示了结果。可以观察到，LPD 比 DCTCP 性能提高约 30%，比 D²TCP 提高大约 20%。为了测试 LPD 在减少流完成时间方面的性能，让每个服务器 S_i 将流发送到随机目的节点，流发送过程持续 2 分钟，重复这个过程 20 次，图4.25 (a) 显示了实验结果。可以看到，与 DCTCP 和 L²DCT 相比，LPD 流平均完成时间减少了约 20% 和 10%。随后测试 LPD 在 Partitions-Aggregate 的性能，首先让根节点服务器 R 向每

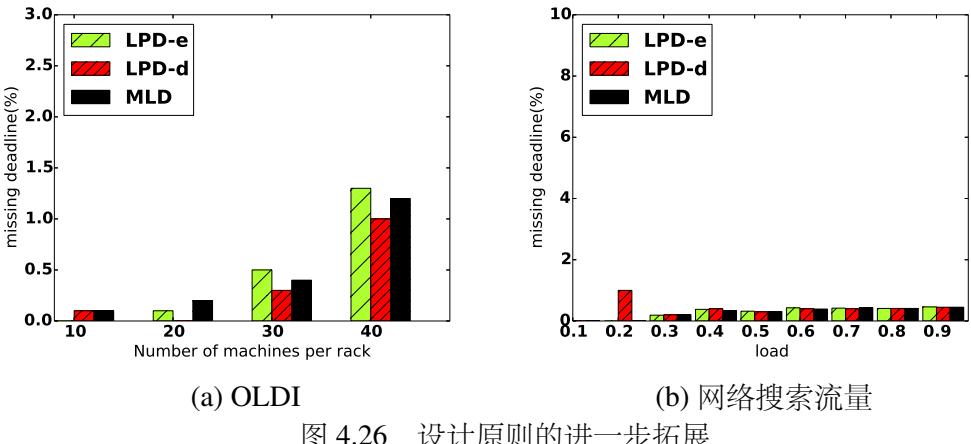


图 4.26 设计原则的进一步拓展

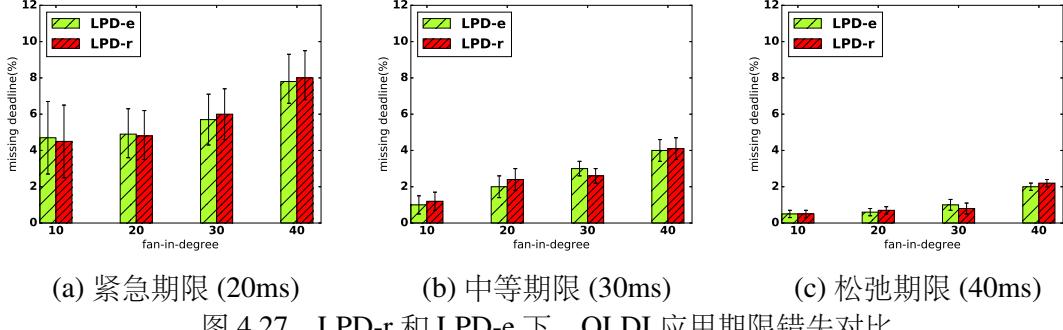


图 4.27 LPD-r 和 LPD-e 下，OLDI 应用期限错失对比

个端节点发送查询请求，在端节点接收到查询请求后，它将发送流大小为 K 的响应数据流。将 K 从 50 KB 变为 200 KB，图4.25 (b) 显示实验结果。可以看到，与 DCTCP 和 L^2DCT 相比，LPD-s 使流平均完成时间分别减少 20% 和 10%。这是因为 LPD 采用具有“越拥塞，越区分”的惩罚函数，因此端节点的拥塞控制机制在负载很重的情形下依然有效。

4.6 对“越拥塞，越区分”原则的讨论

“越拥塞，越区分”这个原则是具有通用性的，本文的策略使用的控制形式：

$$w = \begin{cases} w + (1 - f) & \text{没有拥塞;} \\ w \times (1 - f) & \text{发生拥塞} \end{cases}$$

本部分，使用不同的惩罚函数来测试“越拥塞，越区分”原则的通用性。

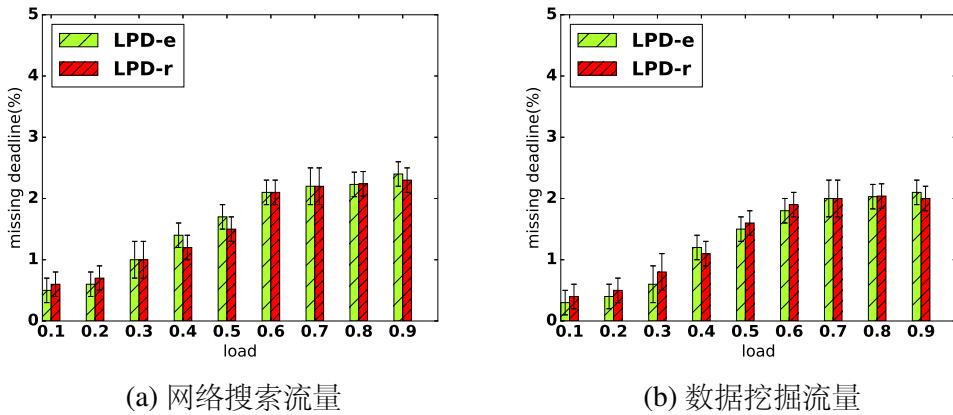


图 4.28 LPD-r 和 LPD-e 在下的网络搜索和数据挖掘流量下错失期限对比，流截止期限在 10ms 到 30ms 之间，超额认购因子为 10

4.6.1 不同的惩罚函数

设 $f=a/d$, 其中 d 是在 D²TCP 中使用的期限因子, 使用 $f=a/d$ 作为惩罚函数, 把这个策略成为 LPD-d。LPD-d 已经用来解决研究动机提出的问题。此外, 引入另一种拥塞控制机制:

$$f = \begin{cases} (2-d)^\alpha & 0 < d < 1 \\ \alpha & d = 1 \\ \alpha * \frac{2-d}{2d-\alpha} & d > 1 \end{cases}$$

可以证明 $\frac{\partial(1-f)}{\partial d} > 0$, 此外 $\frac{\partial^2(1-f)}{\partial \alpha \partial d} > 0$, 此例符合本文提出的“越拥塞, 越区分”的原则, 称之为 MLD。

设置流的期限满足统一分布, 图4.26(a) 描述了 OLDI 应用下 LPD-e, MLD 和 LPD-d 性能对比。图4.26(b) 下网络搜索流量下的结果。图4.26显示不同形式的算法的差异性不大, 策略能够在网络拥塞程度重的情形下使的期限不同的数据流。

4.6.2 使用剩余时间

在4.4节中提出 LPD-e, 使用流程开始时间和截止期限之间的时间间隔。因为, 减少流错失期限的最优策略是最小期限优先策略。使用另一种策略, 使用当前时间和截止期限之间的持续时间作为调整因子, 称之为 LPD-r 策略。

使用的相同参数来评估 LPD-r, 并将其与 LPD-e 进行比较。图4.27描绘了在简单拓扑下 OLDI 应用 LPD-r 与 LPD-e 错失期限情形对比。而图4.28描绘了在 spine-leaf 拓扑中的在网络搜索和数据挖掘下错失期限对比。从实验结果中, 可以看到 LPD-r 和 LPD-e 性能相近, 这再一次证实了“越拥塞, 越区分”原理的通用性, 因此

在实际中，可以使用不同的惩罚函数以及不同的期限因子设计拥塞控制机制。

4.7 本章总结

本章提出“越拥塞，越区分”的原则来设计数据中心拥塞控制机制。根据本原则设计的拥塞控制机制，使的网络即使在严重拥塞的情形下依然可以根据截止期限分配带宽。为了证明这个原则的有效性，本文设计了速率控制机制－负载自适应拥塞控制策略（load proportional differentiation，简称 LPD），并使用不同的网络拓扑和网络流量测试 LPD 的性能。实验发现 LPD 的性能几乎超越了例如 D²TCP, L²DCT 等当前最先进的拥塞控制机制。

尽管 LPD 用来使的更多数据流在截止期限之前完成，原则“越拥塞，越区分”是有通用性，用户可以根据这个原则设计其它的拥塞控制机制。然而，关于 LPD 的收敛性和稳定性，在以后的工作中需要进行进一步的讨论。

第5章 基于流持续时间的传输优化方案

当前，许多应用（如网络搜索和零售）部署在数据中心。由于 TCP 不能满足应用对延迟和吞吐量的需求，因此研究者们提出了许多传输协议（例如，DCTCP, D²TCP, L²DCT）来对 TCP 进行补充。其中，D²TCP 等协议将流的截止期限纳入拥塞窗口调整过程，以保证流在截止期限之前传输完成。在数据中心中，短流常常需要较短延迟，因而，L²DCT 等协议在计算拥塞窗口调整因子时考虑流大小，以此保证短流的吞吐量和延迟。这两种方法在一定的场景下运行良好，但依旧存在一些不足：首先，这两种方法只能减小流错失期限的百分比或者能够减小流平均完成时间，但是不能够同时优化这两个目标。其次，大多数方法都需要预知流的信息（例如截止期限，流大小），但是对于很多应用，流大小以及流截止期限这些信息很难预先得知。因此，在本章中，本文主张使引入流持续时间到拥塞窗口调整的过程中。基于此，本文提出流持续时间速率控制机制（Flow Duration Time Rate Control，简称 FDRC）。本文发现，在不用预先得知流信息的情形下，FDRC 可以减少流错失期限的比例并且能够减小流平均完成时间。本章从理论上分析了 FDRC 的行为，并在 ns-2 和 Linux 内核实现 FDRC。同时实验表明，在几乎所有场景下，FDRC 比 D²TCP 和 L²DCT 的性能都要好。平均来说，FDRC 比基于期限的拥塞控制协议 D²TCP 性能高 30%，比优化流平均完成时间的方法 L²DCT 性能提高大约 10%。

5.1 概述

如今，越来越多对延迟敏感的应用（例如网络搜索，零售）被部署在数据中心网络。数据中心网络（DCNs）需要给应用提供高吞吐，低延迟的服务，此外还需要能够承受高并发等特殊需求。最近，在这些需求中，延迟引起了更多的关注，因为这会影响用户在这些应用中的体验，从而影响收入。事实上，对于数据中心应用，每增加 100ms，可能导致 1% 的收入损失^[16,57]。

特别的，数据中心的流是由长的背景流，对时延敏感的流和带宽敏感流的混合，使用 TCP 不能满足所有流的需求，会导致下面的两个问题。首先，使用 TCP 协议，会导致交换机缓冲区溢出，数据包重传，进而使的网络延迟加大，从而影响用户体验。其次，网络发生拥塞时，TCP 发送端拥塞窗口会减半，导致网络震荡严重，链接利用率低。由于存在 TCP 以上不足，DCTCP^[16] 被提出。DCTCP 采用大多数现代交换机支持的 ECN 标记机制^[16]，首先在交换机上设置一个阈值 K，当

交换机缓冲区队列超过 K 时，发送到交换机的数据包会标记 CE，接收端收到所有的数据包后，如果有被标记 CE 的数据包，那么回复给发送端的 ACK 就会被标记 ECN。最后发送端统计上一个 RTT 中收到的 ACK 被标记的 ECN 的比例，然后计算链路拥塞程度，最后根据拥塞程度调整拥塞窗口。DCTCP 通过使的交换机缓冲区队列维持在较短水平上，从而可以减小排队延迟，同时可以维持高链路利用率。但是，DCTCP 不考虑流的期限问题，它对所有流都公平对待，这样会导致许多对延迟敏感的流错过期限^[23,27]。针对此，业界提出两类方法来弥补 TCP 的不足：

(1) 给流设置显式的期限。例如 D³^[27]，D²TCP^[23] 和 LPD^[57] 这几个方法是从用户态给应用的数据流设置期限，数据流带宽由网络拥塞程度和期限共同决定。截止期限近的数据流得到的带宽比期限远的流得到的带宽多，因此，截止期限近的流和截止期限远的流都能在截止时间之前完成。通过设置不同的截止期限，优化数据流的传输延迟。

(2) 将短流设置为高优先级。在数据中心网络中，短流常常承载交互等信息，因此往往需要低延迟。所以 PDQ^[28]，L²DCT^[24] 和 pFabric^[32] 等试图将短流设置为高优先级，长流设置为低优先级，并试图实现最短流优先策略（Shortest Job First，简称 SJF）。事实上，在单条瓶颈链路的场景下，最短流优先策略是减小流平均完成时间的最优策略。事实上，在多链路的情形下，最短流优先策略也能达到较好的性能^[24,32]。

事实上，这两种方法都试图把紧急的流设置更高的优先级，优化紧急流的延迟。区别在于第一种方法是根据用户期望的延迟设置优先级，第二种是根据流的大小设置优先级。两种方法都需要用户事先得知流信息。具体的，如果使用第一类方法，需要事先得知流的截止时间；如果使用第二类方法，需要事先得知流大小。另外，数据中心的流是延迟敏感流，带宽敏感流和长的背景流的混合，第一类方法可以满足延迟敏感流的需求，第二类方法只是试图降低流的平均完成时间。事实上，这两种方法都不能同时满足延迟敏感流，带宽敏感流和长的背景流的需求，所以这些方法在部署上有一定的局限性。

由于以上提到的缺陷，本文提出流的优先级可以根据流启动的持续时间来设置。流在开始发送时具有最高优先级，随着时间的推移其优先级下降。一方面不需要事先知道流的信息（流的大小或截止时间）；另一方面，这种方法间接给短流设置了更高的优先级，短流经常有期限限制，短流能获得较多的带宽，进而能够在截止时间之前完成，同时优化了流的平均完成时间。基于此，本章提出了基于流持续时间的速率控制算法（Flow Duration Rate Control，简称 FDRC）。本章将 FDRC 实现在 ns-2^[48] 和 Linux 内核 3.2.61 上。实验结果表明 FDRC 的性能比 D²TCP 提高

了30%，比DCTCP提高了2×。本章中主要做了以下的工作：

(1) 提出数据中心基于流持续时间的传输协议(Flow Duration Rate Control, 简称FDRC)。FDRC可以使更多流达到最终期限的限制，同时有助于减少流平均完成时间。FDRC可以在事先不知道流信息的情形下，有效的减少流错过期限数目的百分比。

(2) 在ns-2中评估FDRC，然后用DCTCP, D²TCP, L²DCT, LPD和pFabric与之对比。最后，在linux内核中实现FDRC，并构建小型数据中心测试FDRC，进而评估FDRC的性能。

5.2 相关工作和研究动机

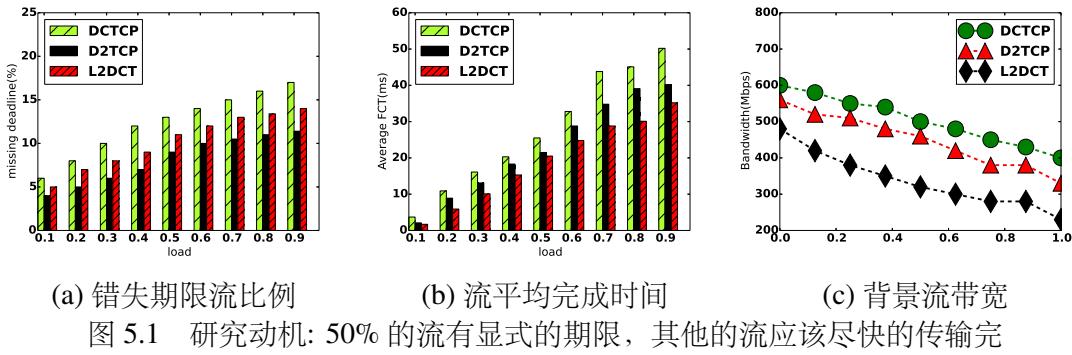
5.2.1 相关工作

业界已经提出一系列方法来保证数据中心流传输延迟，根据这些方法特性可以分为三类，第一类是基于TCP的方法，例如DCTCP^[16], D²TCP^[23]以及LPD^[57]等，这类方法一般借助交换机ECN标记机制，通过ECN机制来评估交换机队列长度，根据网络拥塞程度和流特性(期限，流大小)等进行速率控制；第二类是网络调度方法，例如PDQ^[28], D³^[27]等，这类方法通常采用集中控制的方法，在此系统中，有一个集中控制器，集中控制器需要预先得知网络的情形，根据网络实时拥塞状况对流进行调度；第三类是控制交换机队列的方法，例如pFabric^[32]等，这类方法需要改动交换机队列，在交换机上实现特殊的队列控制机制，通过特殊队列进出方式，实现对流的速率控制。

以上的三类方法，无论是显式设置期限还是隐式的优化流完成时间，都需要预知流的信息(流大小，截止期限，剩余流大小，剩余时间等)或者对网络设备进行修改。但是在数据中心的流是混合了延迟敏感流，带宽敏感流和长的背景流。给流设置截止期限，根据截止期限区分优先级，能够减少流错过期限的比例。根据流大小进行调整，只能够优化流完成时间。但是当前没有一种方法能够同时满足这两者的需求，此外这些方法还需要预先得知流信息。实际需要一种无须知道流的信息就能够优化有截止期限的流和流完成时间。

5.2.2 研究动机

当前速率控制算法，无论是设定显式截止期限或者给短流设置高优先级，优化流平均完成时间，可以使流在截止期限之前完成或减少流平均完成时间，但是不能同时满足两个目标。为了验证此问题，在ns-2平台上，启动10个发送端，1个接收端，它们共同连接到一个交换机上，设置交换机的缓冲区的阈值K=25。实



(a) 错失期限流比例 (b) 流平均完成时间 (c) 背景流带宽

图 5.1 研究动机: 50% 的流有显式的期限, 其他的流应该尽快的传输完

验中的所有链路均设置 1Gbps 的带宽, 在 10 个发送端中, 有 2 个持续发送背景流, 另外 8 个不断发送大小从 100KB 到 1MB 的短流, 将短流量平均分为两组: 对于第一组 (4 个发送端), 期望短流能够在 30ms 之内完成传输 (截止期限为 30ms); 对于第二组 (4 个发送端), 期望流平均完成时间尽可能短。仿真时间持续 10 分钟, 最终结果如图 5.1 所示。

图 5.1 (a) 显示了几种方案下错失期限的流数目对比, 图 5.1 (b) 显示了流平均完成时间的对比。图 5.1 (a) 所示, D²TCP 比 L²DCT 和 DCTCP 性能更好, 图 5.1 (b) 所示, 对于流平均完成时间, L²DCT 性能更好。这是因为使用 D²TCP, 有显式期限的紧急流优先级较高, 从而获得较多的带宽, 因而更有希望在截止时间之前完成。但是, 使用 DCTCP 和 L²DCT, 第一组有期限的流与背景流的优先级相同, 因此很多期限近的流无法获得更多的带宽, 从而会错失截止期限。对于流平均完成时间, L²DCT 比 D²TCP 性能更好, 这是因为在 L²DCT 中, 短流优先级高于长流, 优先完成短流, 因此能够优化流平均完成时间, 但是在实际中并不是所有的短流都有期限, 因而, 对于有期限的流, L²DCT 并不能完全优化, 所以, 很多有期限的流并不能在截止时间之前完成。图 5.1 (c) 显示了背景流的带宽。可以看到, DCTCP 下的背景流的带宽比 D²TCP 与 L²DCT 下背景流带宽高。

从图 5.1 可以得出以下结论: D²TCP 可以减少流错失期限的比例, L²DCT 可以减少流平均完成时间。但是, 无论 D²TCP 还是 L²DCT 均不能同时满足这两个目标。这是因为 D²TCP 使用截止期限 (deadline) 作为速率控制因子, 所以截止期限近的流将比截止期限远的流有更大的带宽。但是 D²TCP 无法优化流平均完成时间。L²DCT 使用流大小作为速率控制因子, 因此小流将比长流有更高的平均带宽。但是, 在数据中心, 并不是所有短流都有明确的截止期限 (deadline), 有些流只需要尽可能快的传输 (例如小的请求)。L²DCT 将导致流平均完成时间较短, 但是对于有显式期限的数据流会错失期限。

由于数据中心的流是有紧急期限的流, 带宽敏感的流 (希望流完成时间短) 的混合, 所以需要一个协议来同时满足这两种流的需求。当前无论 D²TCP 或者

L^2DCT 均不能达到此目的，它们只能优化有期限的流或者优化流完成时间，而无法同时满足两种需求。因此，建议把流持续时间作为速率调整因子引入到流的速率控制中。开始启动时，流具有最高优先级，随着时间流逝，流优先级下降。引入流持续时间是一个很好的选择，因为一方面这会使短流具有较高的平均带宽，所以流有较短的平均完成时间；另一方面，在固定时间阈值内的流将具有较大的平均带宽，对于有期限的流也是一种较好的优化方法。

5.3 FDRC 策略

5.3.1 基于流持续时间的速率控制策略

Algorithm 1: 拥塞窗口因子计算算法

Input: 流启动时刻 $time_start$, 当前时刻 $time_now$

Output: 窗口调整因子 d

```

1 durtme = time_now - time_start;
2 if durtme ≤ threshold_tight then
3   d = DMIN;
4 else if durtme ≥ threshold_lax then
5   d = DMAX;
6 else
7   d=DMIN +  $\frac{(\textit{durtme}-\textit{threshold\_tight})*(\textit{DMAX}-\textit{DMIN})}{\textit{threshold\_lax}-\textit{threshold\_tight}}$ 
8 return d;
```

将流持续时间引入拥塞窗口控制，基于此，本文提出了基于流持续时间的策略（Flow Duration Rate Control，简称 FDRC）。算法1展示了计算期限因子的步骤。算法1计算期限因子，需要四个参数： $threshold_tight$ 和 $threshold_lax$ 是时间阈值， $DMAX$ 和 $DMIN$ 是期限因子的上限和下限，当流的持续时间大于 $threshold_lax$ 时，期限因子 d 被设置为 $DMIN$ ，当流持续时间小于 $threshold_tight$ ，期限因子 d 被设置为 $DMAX$ 。当持续时间短于 $threshold_tight$ 时，流保持最高优先级。如果流持续时间大于 $threshold_lax$ ，则流保持最低优先级。对于 $threshold_tight$ 和 $threshold_lax$ 之间的流，优先级随着时间的推移而下降。算法1在每个 RTT 执行一次。随着时间的推移，流优先级下降，因此，长的背景流的带宽平均值小于短流的带宽，流完成时间长。

5.3.2 FDRC 协议

FDRC 基于 DCTCP，在交换机端使用 ECN 标记模式。交换机预先设置一个标记的阈值 K 。当队列长度大于交换机的阈值 K 时，到达的数据包将被标记 CE。然后，接收端将检查来的数据包是否被标记，如果发送过来的数据包有 CE 标记，则接收端回复的 ACK 中有 ECN 标志；如果没有 CE 标记，那么接收端收到的 ACK 中没有 ECN 标志。发送方每个 RTT 时间段，计算一下所有 ACK 中被标记 ECN 的比例，得到拥塞程度 α ：

$$\alpha = (1 - g) \times \alpha + g \times F \quad (5-1)$$

其中， F 是上一个 RTT 被标记 ECN 的比例。 g 是滑动平均的因子，拥塞程度 α 反应的是这一段时间内的拥塞情况。根据计算的拥塞程度 α ，得到 FDRC 拥塞窗口计算方法：

$$f = \alpha \times d \quad (5-2)$$

根据^[57]，拥塞窗口的变化的加法增加部分和乘法减少部分可以是相同的。根据此，改动滑动窗口的变化如下：

$$p = \begin{cases} w + (1 - f) \\ w * (1 - f) \end{cases} \quad (5-3)$$

5.3.3 FDRC 分析

在本部分，从理论上分析 FDRC 的性能，建立两个模型来分析 FDRC 的性能：锯齿模型和 FDRC-fluid 模型。锯齿模型是一个粗粒度的模型，FDRC-fluid 模型是细粒度的模型。本文使用锯齿模型用来分析 FDRC 速率和参数的关系，使用流模型，分析 FDRC 的参数。

5.3.3.1 FDRC 锯齿模型

本部分建立锯齿模型^[57]来分析 FDRC 的带宽。 R_1 和 R_2 表示 $flow_1$ 和 $flow_2$ 的速率。假设两条流的拥塞窗口从 w_{min} 增加到 w_{max} ，有 $w^{av} = \frac{w_{min} + w_{max}}{2}$ ，其中 w^{av} 表示的是平均窗口大小。流的拥塞窗口变化是有固定周期的锯齿，滑动窗口从 w_{min} 变化到 w_{max} ，需要 N 个 RTT。根据 (5-3)，滑动窗口每个 RTT 增加 $(1 - f)$ ，

因此得到最大的滑动窗口和最小滑动窗口的关系: $w_{max} = w_{min} + N \times (1 - f)$ 。当出现拥塞时候, 窗口减小到 w_{min} , 根据 (5-3), 有 $w_{min} = w_{max} * (1 - f)$ 。令 w_1 是 flow₁ 的最大窗口, w_2 是 flow₂ 的最大窗口。根据平均滑动窗口, 计算两条流的平均发送速率比:

$$\frac{R_1}{R_2} = \frac{w_1^{av}}{w_2^{av}} \quad (5-4)$$

而 $w^{av} = \frac{w_{min} + w_{max}}{2}$, $w_{max} = w_{min} + N \times (1 - f)$, 带入 (5-4), 得到二者的速率比:

$$\begin{aligned} \frac{R_1}{R_2} &= \frac{(2 - f_1) \times w_1}{(2 - f_2) \times w_2} \\ &= \frac{2 - \alpha d_1}{2 - \alpha d_2} \times \frac{1 - \alpha d_1}{1 - \alpha d_2} \times \frac{d_2}{d_1} \end{aligned}$$

当 d_1 和 d_2 很小, 有 $\frac{2 - \alpha d_1}{2 - \alpha d_2} \approx 1$, $\frac{1 - \alpha d_1}{1 - \alpha d_2} \approx 1$, 因此, 可以得到:

$$\frac{R_1}{R_2} \approx \frac{d_2}{d_1} \quad (5-5)$$

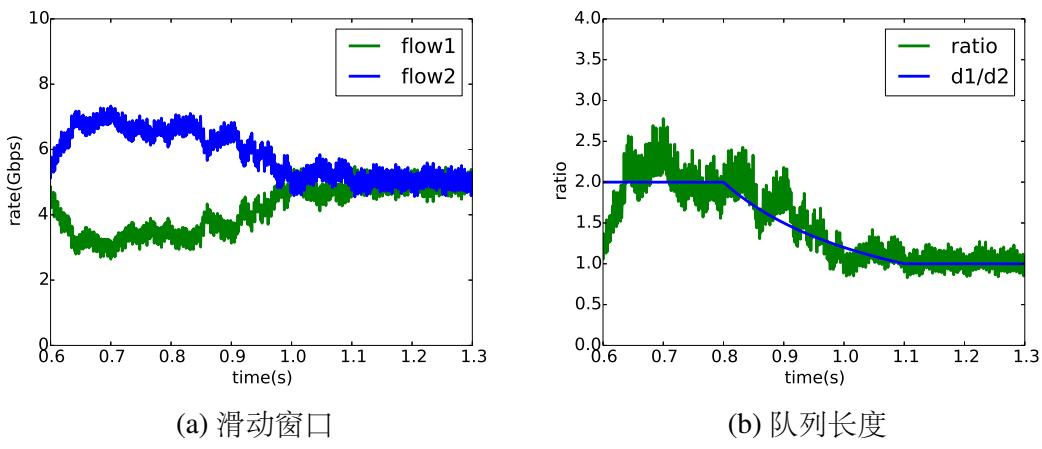


图 5.2 (a) 显示了两条流带宽. (b) 显示二者的速率比和比例因子

从 (5-5) 得知, 对于两条流的窗口调整因子很小的场景, 两条流的传输速率和窗口调整因子成反比。为了测试模型的准确度, 分别在 $t=0s$ 和 $t=0.5s$ 启动两条长流 (其中 flow₁ 的启动时间 $start_time = 0s$, flow₂ 的启动时间 $start_time = 0.5s$)。设定阈值 $threshold_lax = 0.6s$, 阈值 $threshold_tight = 0.3s$, $DMIN = 0.05$, $DMAX = 0$ 。当阈值 $threshold_lax = 0.6s$ 时, flow₁ 从 $t = 0s$ 开始, 所以在 $t = 0.6s$ 之后, flow₁ 的窗口调整因子 $d = DMIN$ 。flow₂ 从 $t = 0.5s$ 开始, 因此从 $t = 0.5s$ 到 $t = 0.8s$, 窗

口调整因子具有最大因子 $DMAX$ 。 $t = 0.8s$ 后，窗口调整因子开始下降。从 $t = 1.1s$ 后， $flow_1$ 和 $flow_2$ 具有相同的调整因子。

图5.2(a) 显示了两条流速率情形。从图5.2(a) 可以看出，从 $t = 0.6s$ 到 $t = 1.1s$ ， $flow_1$ 带宽大于 $flow_2$ ， $t = 1.1s$ 后，两条流的带宽几乎相等。图5.2(b) 显示了 $flow_1$ 和 $flow_2$ 的带宽比和因子比。从 $t = 0.6s$ 到 $t = 0.8s$ 可以得知，两条流的因子比是 $DMAX/DMIN = 2$ ，两条流的速率在此附近波动。从 $t = 0.8s$ 到 $t = 1.1s$ ，两条流的窗口调整因子的比开始下降，两条流的速率比也在下降。在 $t = 1.1s$ 之后，两条流的调整因子比为 1，两条流的带宽基本相同。

5.3.3.2 FDRC 流模型

假设 N 个并发的流通过容量为 C pkts / sec 的瓶颈链路传递给同一个接收者节点，假设传输延迟为 pd ，设置 (3-1) 中 $f_1(\alpha) = f_2(\alpha) = \alpha \times d$ ，同时把 $f_1(\alpha)$ 和 $f_2(\alpha)$ 带入 (3-2)~(3-4)，并且改动 (3-5) 中数据包标记的方法，并且增加 α 标记方法，那么 FDRC-fluid 模型可以描述为：

$$\frac{d\alpha_i}{dt} = \frac{g}{R(t)}(p(t - R^*) - \alpha_i(t)) \quad (5-6)$$

$$\hat{p}(t) = 1_{\hat{q}(t)>1} \quad (5-7)$$

$$\frac{dq}{dt} = \sum_{i=1}^N \frac{w_i(t)}{R(t)} - C \quad (5-8)$$

$$\frac{dw_i}{dt} = \frac{1 - \alpha_i(t) \times d_i(t)}{R(t)} - \frac{w_i(t) \times \alpha_i(t) \times d_i(t)}{R(t)} p(t - R^*) \quad (5-9)$$

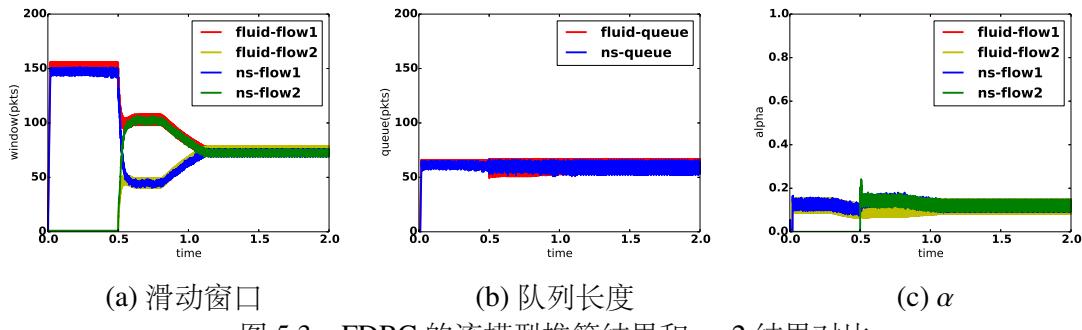


图 5.3 FDRC 的流模型推算结果和 ns-2 结果对比

等式(5-7)和等式(5-8)描述了交换机标记数据包的过程。等式(5-8)描述了队列的长度，等式(5-9)描述发送端窗口长度。为了验证的模型，启动了两条长流($flow_1$

的启动时间 $time_start = 0s$, $flow_2$ 的启动时间 $time_start = 0.5s$)。 $DMAX = 1$, $DMIN = 0.5$, 设置两个阈值: $threshold_tight = 0.3s$, $threshold_lax = 0.6s$ 。对于持续时间小于 $0.3s$ 的流, 具有最大的调整因子值 $DMAX$, 对于持续时间大于 $0.6s$ 的流, 具有最小的调整因子值 $DMIN$ 。对于持续时间在 $0.3s$ 到 $0.6s$ 之间的数据流, 调整因子在不断减小。为了验证模型, 使用 FDRC 模型来预测拥塞窗口并将 FDRC-fluid 模型推算结果与 ns-2 仿真结果进行比较。

图5.3(a) 描述了拥塞窗口的变化。可以看到, 起初只有 $flow_1$ 开始, 所以 $flow_1$ 占据了所有链路带宽, 此时滑动窗口为 150 MSS。在 $t = 0.5s$ 时, $flow_2$ 启动。在 $t = 0.5s$ 时, $flow_2$ 的因子大于 $flow_1$, 所以 $flow_2$ 拥有更大的拥塞窗口。在 $t = 1.1s$ 后, 它们具有相同的拥塞窗口控制因子, 因此二者的滑动窗口开始逐渐相同。图5.3(b) 显示了队列大小, 图5.3(c) 显示了流的拥塞程度 α 。可以看到, 对于流模型的队列长度和流的拥塞程度 α , 仿真结果和模拟结果基本相似。

5.3.3.3 FDRC 参数选取

实际当中使用 FDRC 协议, 有 4 个全局参数是必需的。首先, 两个时间阈值 $threshold_tight$ 和 $threshold_lax$ 可以根据实际应用的传输期限进行设置。例如, 在数据中心, 数据流传输有截止期限, 截止期限常常是一个时间范围, 可以将 $threshold_tight$ 设置为期限的下界, 同时将 $threshold_lax$ 设置为期限的上界。分析如何设置 $DMIN$ 和 $DMAX$ 。首先, 根据 FDRC-fluid 模型, 得到不动点 $(w_i, \alpha_i, q) = (\frac{1}{d_i} - 1, 1, \sum_{i=1}^N \frac{1}{d_i} - N - C \times pd)$ 。不动点的物理含义是, 所有发送到交换机的数据包都被标记 CE, 此时发送者窗口处于固定状态, 不再发生变化。在实际中, 交换机队列不可能太大, 因为较长的队列会导致较长的排队延迟。在 FDRC 系统中, 队列长度应该小于 $K + N$, 所以可以得到:

$$\sum_{i=1}^N \frac{1}{d_i} - N - C \times pd \leq K + N \quad (5-10)$$

因为, 在 FDRC 中, 窗口调整因子 d 在 $DMIN$ 和 $DMAX$ 之间波动, 因此对于 d 取得 $DMIN$ 时, 可以得到:

$$N \times \frac{1}{DMIN} - N - C \times pd \leq K + N \quad (5-11)$$

因此, 得到:

$$DMIN \geq \frac{N}{2 \times N + C \times pd + K} \quad (5-12)$$

从(5-3), 得知拥塞窗口控制函数 $f < 1$, 当 $\alpha \times d$ 到达 1 时, 可以得到 $d < 1$, 因此 $DMAX$ 和 $DMIN$ 应该小于 1。 $DMIN$ 和 $DMAX$ 应该满足:

$$\frac{N}{2 \times N + C \times pd + K} \leq DMIN < DMAX \leq 1 \quad (5-13)$$

在数据中心中, 端和端之间的并发数往往小于 100^[16], 假定 $C = 10Gbps$, $pd = 100\mu s$, K 设定为 64, 数据包的大小设定为 1500B, 因此可以得到:

$$0.29 \leq DMIN < DMAX \leq 1 \quad (5-14)$$

5.4 实验验证

在本节中, 在 ns-2 的平台和 Linux 真实测试环境下对 FDRC 的性能进行测试。首先在仿真平台上, 用实验发现 FDRC 可以解决动机部分 5.2.2 提出的问题。之后, 验证 FDRC 对流完成时间的优化, 实验结果发现, 对于流完成时间, FDRC 比 DCTCP 降低了约 50%。最后, 在小型数据中心测试平台上评估 FDRC 的性能。

5.4.1 解决研究动机提出的问题

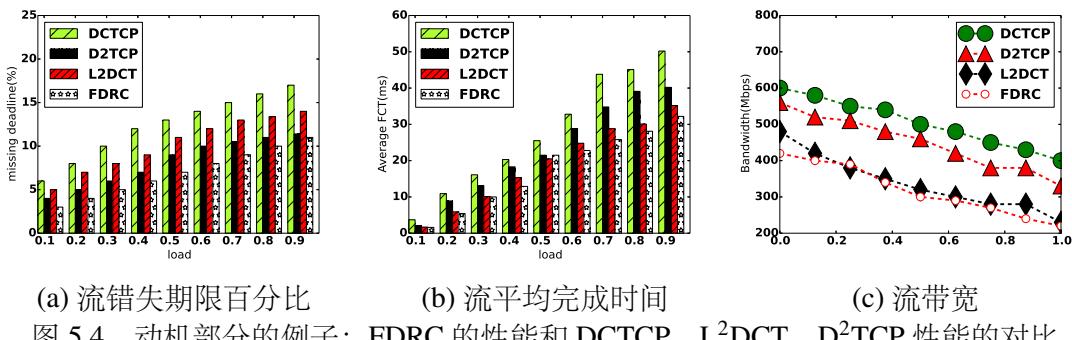


图 5.4 动机部分的例子: FDRC 的性能和 DCTCP, L²DCT, D²TCP 性能的对比

在这一节中, 首先解决动机部分 5.2.2 提出的问题。对于 FDRC, 因为数据中心, 流的期限通常在 10ms 至 30ms 之间, 因此在实验中, 设置阈值 $threshold_lax=30ms$, 阈值 $threshold_tight=10ms$ 。根据 (5-14), 可以设置 $DMAX = 0.8$, $DMIN = 0.3$ 。

对于 DCTCP, D²TCP, L²DCT, 以及其他实验参数和实验环境, 使用和5.2.2相同的配置进行实验。

最终实验结果如图5.4所示。从图5.4(a)看到, 总体上, 几乎在所有情况下, 使用 FDRC 时, 流错失期限的比例比 D²TCP, L²DCT, DCTCP 更少。相对于 DCTCP, FDRC 流错失期限的比例平均少 30%; 相对于 D²TCP, FDRC 流错失期限的比例平均少 20%; 相对于 L²DCT, FDRC 流错失期限的比例平均少 10%。图5.4(b)反应的是流平均完成时间对比结果, 从总计看, 使用 FDRC 时, 对流平均完成时间的优化是最好的。对于流平均完成时间, FDRC 比 DCTCP 平均小 30%, 相比于 D²TCP, FDRC 性能好 20%, 相比于 L²DCT, FDRC 性能好 10%。图5.4(c)反映的是背景流带宽情形, 可以发现 FDRC 获取带宽的能力和 L²DCT 基本相同, 比 D²TCP 和 DCTCP 要强。

分析 FDRC 能同时优化带有期限的数据流和优化流完成时间的原因。首先, FDRC 使用流持续时间作为滑动窗口的控制因子, 在实验中, 设置阈值 *threshold_lax*=30ms, 阈值 *threshold_tight*=10ms, 所以期限在 30ms 以内获得的平均带宽高, 紧急流将比宽松的流更高的带宽, 所以更少的流将错过最后期限。由于刚开始启动时, 流的优先级最高, 随着时间的推移而下降, 所以短流的平均带宽比长流的平均带宽要大, 因而, FDRC 近似实现短流优先策略, 流平均完成时间小。

5.4.2 真实流量和拓扑下仿真

本部分用真实的数据中心流量和复杂的数据中心拓扑来测试 FDRC 的性能。在 ns-2 中构建如图5.5所示的 Spine-Leaf 拓扑。在 ns-2 的仿真实验中, 启动 144 个服务器, 9 个叶子交换机和 4 个主干交换机, 其中 9 个叶子节点交换机和 4 个主干交换机相连接。通过主干交换机 (4 跳) 的端到端往返延迟是 14.6us, 整个叶交换机 (2 跳) 的端到端行程延迟是 13.3us。其中, 额外设置 10us 在终端服务器上, 这模拟了服务器的计算时间。所有实验参数都与 pFabric^[32] 设置的参数相同。

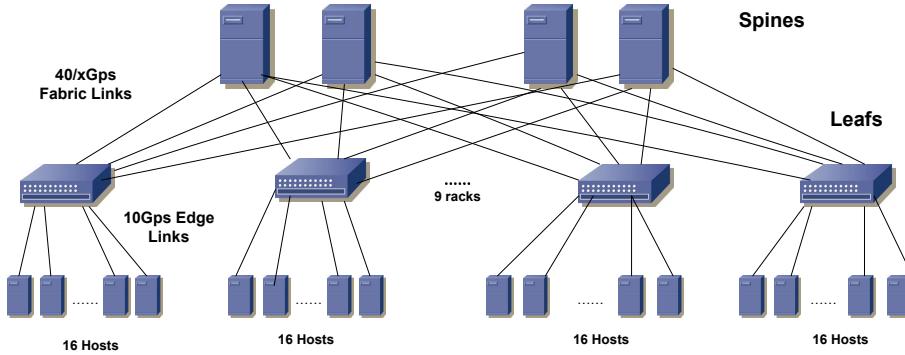


图 5.5 ns-2 建立的数据中心拓扑

数据中心流量特征。使用两种真实数据中心应用的流进行测试：网络搜索流和数据挖掘流^[16,32,57]。对于网络搜索应用，70% 的流小于 1MB，超过 50% 的流在 100KB 和 1MB 之间。对于数据挖掘应用，超过 80% 的流的小于 10KB，超过 90% 的数据量来自 4% 的长流 (> 1MB)。在仿真实验中，本文使用不同的流到达速率来模拟不同的网络负载。为了提高网络资源利用率，因而当前数据中心网络采取超额认购，所以在下面实验中，改变叶子交换机和主干交换机之间的超额认购因子 x 来模拟此。

性能指标。数据中心的数据流是有期限的流，带宽敏感的流，和需要高带宽的背景流的混合。对于有显式截止期限的数据流，选择错过截止期限的流的百分比作为衡量策略针对这类数据流性能的指标。对于短流，考虑流平均完成时间。将 FDRC 与最先进的期限感知拥塞控制算法 LPD-e^[57]，D²TCP^[23] 和对期限不敏感的协议 DCTCP^[16] 进行比较。对于未知截止期限的流，考虑流平均完成时间，并将 FDRC 与 pFabric^[32]，DCTCP^[16] 和 L²DCT^[24] 进行比较。

FDRC 的参数选择。FDRC 需要四个重要的参数。根据^[27]，数据中心的流的期限在 10ms 到 30ms 之间来回震荡，所以选择 $threshold_lax = 30\text{ms}$, $threshold_tight = 10\text{ms}$ 。因子边界 $DMAX$ 和 $DMIN$ 可以根据 (5-14) 设置。在仿真实验中， $DMAX$ 是 0.8， $DMIN$ 是 0.3。

5.4.2.1 时延敏感流优化

在数据中心，网络搜索等时延敏感应用的流应在截止期限前传输完成，如果没有在截止期限前发送完毕，那么会影响应用的性能。在本节，测试了 FDRC 对有截止期限流的优化。实验中，假设流的截止期限服从统一分布，流的到达时间服从泊松分布，此外流的发送端和接收端都是随机选取的。在数据中心，为了提高网络资源利用率，链路存在超额认购，因此，变换叶子节点交换机和主干交换机链路的超额认购因子 x ，并重复各组实验 100 次。在每组实验得到每组实验的最大

值, 最小值, 和平均值, 并在每组的 error bar 上进行标记。同时, 在每组实验中, 使用 DCTCP, D^2 TCP, LPD-e 与 FDRC 作对比。

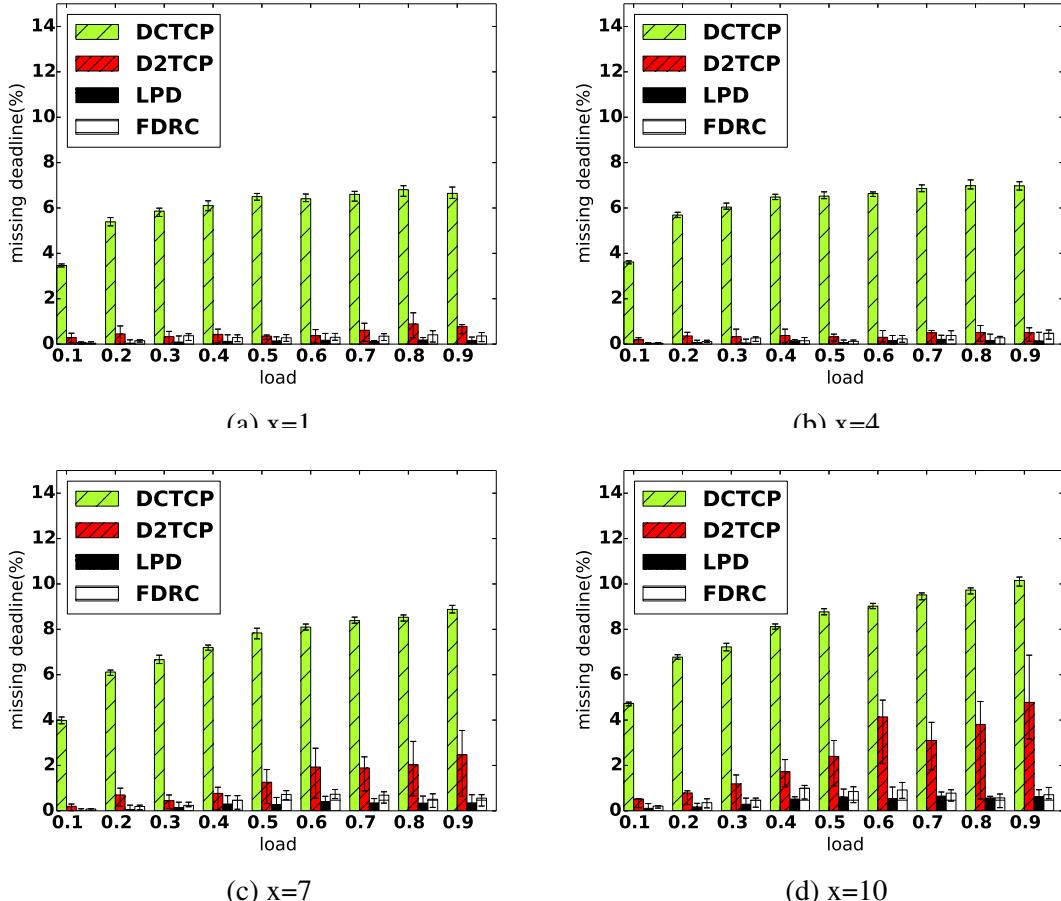


图 5.6 使用网络搜索应用下的 FDRC, LPD, D^2 TCP 之间错失期限的对比情形对比, 流的期限是在 10ms 到 30ms 之间的统一分布

图5.6表示的是网络搜索应用下的实验情况。总体上, FDRC 比 D^2 TCP 和 DCTCP 性能好。图5.6(a)显示的是超额认购系数 $x=1$ 时, 错失的流数目对比。可以发现当网络负载从 0.1 到 0.9 时, 使用 DCTCP 协议, 错失期限流的数目在 3%~7%。而使用 D^2 TCP, 大约有 1%~2% 的数据流错失期限。使用 FDRC, 有 0.2%~1% 的数据流错失期限。使用 LPD, 大约有少于 0.5% 的数据流错失期限。图5.6(b)显示的是超额认购系数 $x=4$ 时, 错失期限的流场景对比。当网络负载从 0.1 到 0.9 时, 使用 DCTCP 协议时, 错失期限的流数目在 3.5%~7.5%, D^2 TCP 大约有 1.5%~2.5% 的数据流错失期限。使用 FDRC, 有 0.7%~1.5% 的数据流错失期限。LPD 下有少于 1% 的数据流错失期限。图5.6(c)显示的是超额认购系数 $x=7$ 时, 错失期限的情形对比。当网络负载从 0.1 到 0.9 时候, 使用 DCTCP 协议时, 错失期限的流的数目大约在 4%~8%。而使用 D^2 TCP 性能要提高一些, 大约有 1%~3% 的数据流错失

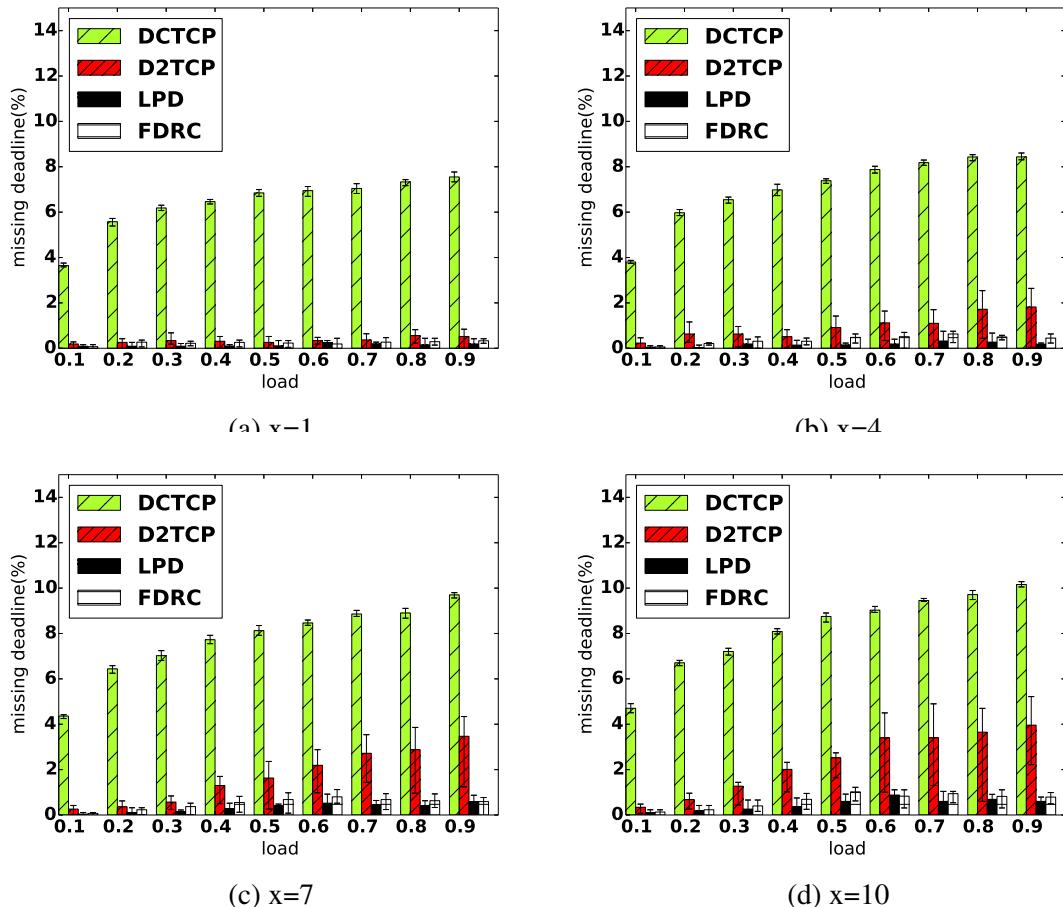


图 5.7 使用数据挖掘应用下的 FDRC, LPD, D²TCP 之间错失期限的对比情形对比, 流的期限是在 10ms 到 30ms 之间的统一分布

期限。使用 FDRC, 大约有 0.3%~2% 比例的数据流错失期限。使用 LPD, 大约有少于 1.5% 的数据流错失期限。图5.6(d)显示的是超额认购系数 $x=10$ 时, 错失期限的对比。当网络负载从 0.1 到 0.9 时候, 使用 DCTCP 协议时, 错失期限的流的数目大约在 3.5%~9%。而使用 D²TCP 性能要提高一些, 大约有 1.5%~4% 的数据流错失期限。使用 FDRC, 大约有 0.7%~3% 比例的数据流错失期限。使用 LPD, 大约有少于 2% 的数据流错失期限。平均而言, 对于网络搜索流而言, FDRC 的流错失期限的比例是 DCTCP 的 1/5, 是 D²TCP 的的 1/2。FDRC 的性能仅次于 LPD。

图5.8表示的是数据挖掘流下的实验情况。总的来看, FDRC 依然是性能最好的。图5.8(a) 显示的是超额认购系数 $x=1$ 时, 错失期限的情形对比。可以发现当网络负载从 0.1 到 0.9 时候, 使用 DCTCP 协议时, 错失期限的流的数目大约在 3.5%~6.5%。而使用 D²TCP 性能要提高一些, 大约有 0.3%~0.8% 的数据流错失期限。使用 FDRC, 大约有 0.1%~0.3% 比例的数据流错失期限。使用 LPD, 大约有少于 0.3% 的数据流错失期限。图5.8(b) 显示的是超额认购系数 $x=4$ 时, 错失期限的

情形对比。当网络负载从 0.1 到 0.9 时候，使用 DCTCP 协议时，错失期限的流的数目大约在 4%~8%。而使用 D²TCP 性能要提高一些，大约有 0.5%~2% 的数据流错失期限。使用 FDRC，大约有 0.3%~1% 比例的数据流错失期限。使用 LPD，大约有少于 0.5% 的数据流错失期限。图5.8(c) 显示的是超额认购系数 x=7 时，错失期限的情形对比。当网络负载从 0.1 到 0.9 时候，使用 DCTCP 协议时，错失期限的流的数目大约在 4%~8.5%。而使用 D²TCP 性能要提高一些，大约有 1%~3.5% 的数据流错失期限。使用 FDRC，大约有 0.4%~0.9% 比例的数据流错失期限。使用 LPD，大约有少于 0.7% 的数据流错失期限。图5.8(d) 显示的是超额认购系数 x=10 时，错失期限的情形对比。当网络负载从 0.1 到 0.9 时候，使用 DCTCP 协议时，错失期限的流的数目大约在 4.5%~9%。而使用 D²TCP 性能要提高一些，大约有 0.5%~4% 的数据流错失期限。使用 FDRC，大约有 0.4%~1% 比例的数据流错失期限。使用 LPD，大约有少于 1% 的数据流错失期限。

平均而言，对于数据挖掘场景下，FDRC 的表现和 web 搜索场景下基本相似。使用 FDRC 导致流错失期限的比例是 DCTCP 的 1/5，是 D²TCP 的的 1/2。FDRC 的性能仅次于 LPD。认为，虽然 FDRC 的性能不及 LPD，但是 LPD 需要预先得知流的大小和期限，而在数据中心中，很多应用数据流期限和流大小是很难预先得知的，因此 FDRC 的应用范围要比 LPD 广泛，尽管 FDRC 有一些性能损失。

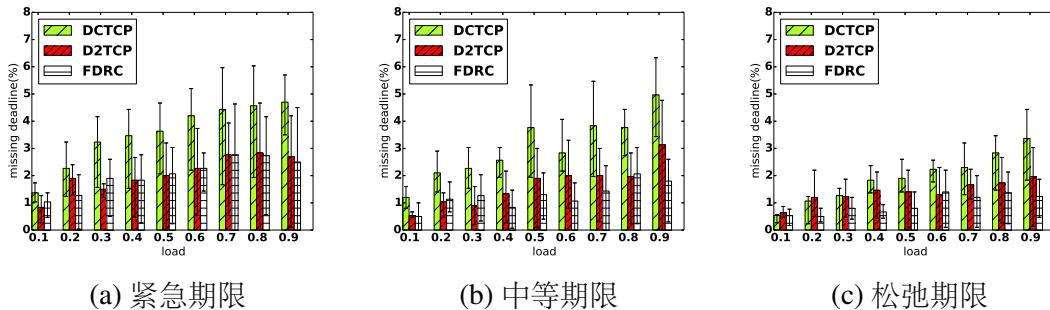


图 5.8 网络搜索应用的场景下，FDRC 和 LPD, D²TCP, DCTCP 错失期限对比。流的期限服从指数分布

在数据中心中，并不是所有的流都存在明确期限，数据中心的数据流往往是有期限的流，没有期限的短流，背景流的混合。在本组实验中，把网络搜索流平均分成两组：第一组，设置显示的期限；第二组，是背景流，没有显式的期限。在仿真实验中，设置三组期限：紧急期限 (10ms)，中等期限 (20ms)，松弛期限 (30ms)。设定所有流的到达服从泊松分布，数据流的源端和目的端都是随机选择的。图5.8是实验结果。

图5.8(a) 显示的是紧急期限 (10ms) 情形下流错失期限的情形对比。发现使用

DCTCP 大约有 1%~6% 的流错失期限。使用 D²TCP 大约有 1%~3% 的流错失期限。使用 FDRC 大约有 1%~2.5% 的流错失期限。FDRC 比 DCTCP 和 D²TCP 性能提高 1× 和 10%。图5.8(b) 显示的是中等期限 (20ms) 情形下流错失期限的情形对比。发现使用 DCTCP 大约有 0.5%~5% 的流错失期限。使用 D²TCP 大约有 1%~2.5% 的流错失期限。使用 FDRC 大约有 1%~2% 的流错失期限。FDRC 比 DCTCP 和 D²TCP 性能提高 2× 和 20%。图5.8(c) 显示的是松弛期限 (30ms) 情形下流错失期限的情形对比。发现使用 DCTCP 大约有 1%~4% 的流错失期限。使用 D²TCP 大约有 0.5%~1.6% 的流错失期限。使用 FDRC 大约有 0.5%~1% 的流错失期限。FDRC 比 DCTCP 和 D²TCP 性能提高 4× 和 40%。平均来看, 对于有期限的流的优化, FDRC 的性能比 DCTCP 提高大约 4 倍, 比 D²TCP 的性能提高大约 30%。

5.4.2.2 流平均完成时间对比

在多瓶颈链路场景下, 最小化流平均完成时间和多染色问题是等价的^[58]。在单条链路上, 最短流优先 (Shortest-Flow-First, 简称 SFF) 策略是最优策略。FDRC 近似的实现了最短流优先策略, 因此对流平均时间的优化较好。

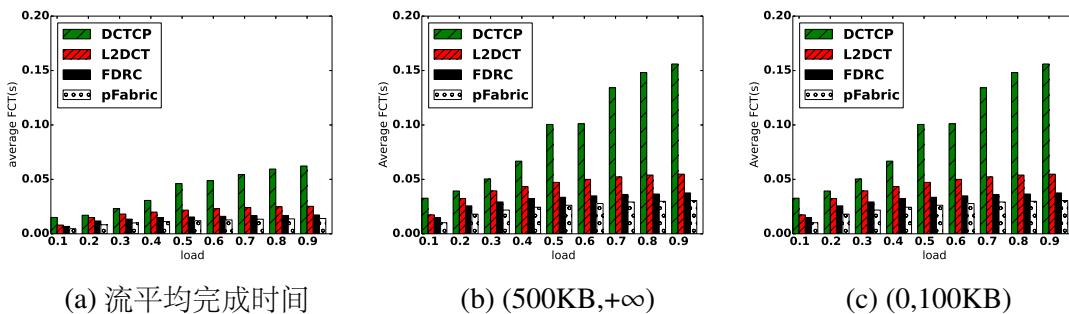


图 5.9 网络搜索应用的场景下, FDRC, DCTCP, L²DCT, pFabric 下的流平均完成时间对比。图 (a) 展示的是流平均完成时间对比情况; 图 (b) 显示的是长流的流平均完成时间; 图 (c) 显示的是短流的平均完成时间

图5.9显示的是网络搜索应用下的流平均完成时间结果对比。图5.9(a) 展示的是流平均完成时间的整体情形。可以看到当网络的负载从 0.1~0.9 时, 使用 DCTCP, 流的平均完成时间 0.01s~0.05s。优化流平均完成时间的方法 L²DCT, 性能要更好, 使用 L²DCT, 流的平均完成时间在 0.01s~0.03s。总体看 L²DCT 的性能比 DCTCP 要提高 1×。使用 FDRC, 流的平均完成时间要更加好, 当网络负载从 0.1~0.9 时, 流的平均完成时间在 0.005s~0.025s。pFabric 是当前的优化流平均完成时间的最优策略, pFabric 的流平均完成时间在 0.003s~0.023s 之间。pFabric 的性能比 FDRC 要高 20%。FDRC 的性能比 L²DCT 提高 30%。图5.9(b) 展示的是长流平均完成时

间的结果。可以看到当网络负载从 0.1~0.9 时，使用 DCTCP 流平均完成时间在 0.03s~0.15s。针对流平均完成时间的方法 L²DCT，表现的要好一些，使用 L²DCT，流的平均完成时间在 0.02s~0.05s 总体看 L²DCT 的性能比 DCTCP 要提高 1×。pFabric 是当前的优化流平均完成时间的最优策略，pFabric 的流平均完成时间在 0.005s~0.02s 之间。pFabric 的性能比 FDRC 要高 20%。FDRC 的性能比 L²DCT 提高 25%。图5.9(c) 展示的是短流平均完成时间的整体情况。可以看到当网络的负载从 0.1~0.9 时，使用 DCTCP 的流的平均完成时间在 0.01s~0.05s。使用 L²DCT，流的平均完成时间在 0.01s~0.03s，使用 FDRC，流的平均完成时间要更加好。pFabric 的流平均完成时间在 0.003s~0.023s 之间。pFabric 的性能比 FDRC 要高 20%。而 FDRC 的性能比 L²DCT 提高 30%。

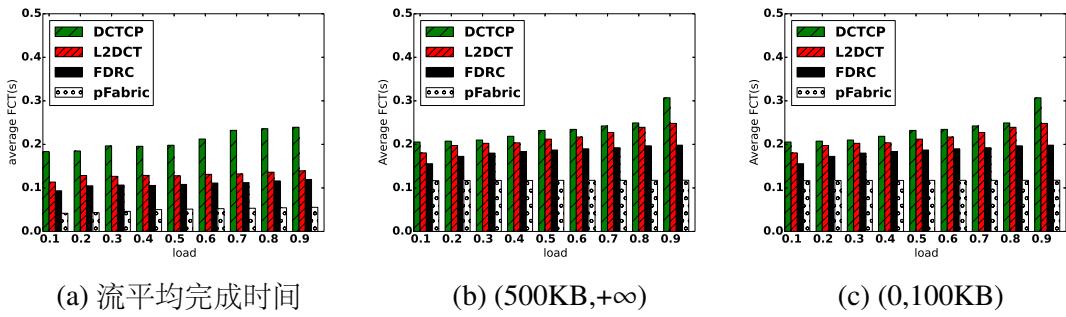


图 5.10 数据挖掘应用的场景下，FDRC, DCTCP, L²DCT, pFabric 下的流平均完成时间对比。图(a)展示的是平均的流完成时间对比情况；图(b)显示的是流大小大于 500KB 时的情形；图(c)显示的是短流的平均完成时间

图5.10显示的是数据挖掘应用下的流平均完成时间的对比。图5.10(a) 展示的是流平均完成时间的整体情况。可以看到当网络的负载从 0.1~0.9 时，使用 DCTCP 的流的平均完成时间在 0.18s~0.25s。针对流平均完成时间的方法 L²DCT，表现的要好一些，使用 L²DCT，流的平均完成时间在 0.11s~0.15s 当网络负载从 0.1~0.9 时，使用 FDRC 的流平均完成时间在 0.008s~0.14s。使用 pFabric 的流平均完成时间在 0.05s~0.08 之间。pFabric 的性能比 FDRC 要高 30%。FDRC 的性能比 L²DCT 提高 12%。图5.10(b) 展示的是长流平均完成时间的结果。可以看到当网络的负载从 0.1~0.9 时，使用 DCTCP 的流的平均完成时间在 0.2s~0.3s。针对流平均完成时间的方法 L²DCT，表现的要好，使用 L²DCT，流的平均完成时间在 0.18s~0.22s 使用 FDRC，流的平均完成时间在 0.14s~0.18s，使用 pFabric 的流平均完成时间在 0.11s~0.15s 之间。FDRC 的性能比 L²DCT 提高 15%。图5.10(c) 展示的是短流平均完成时间的情形，结果与图5.10(a) 以及图5.10(b) 基本相同。

因此总体看 FDRC 的性能比 DCTCP 和 L²DCT 要高，但是不如 pFabric。但是

pFabric 是调度的方法，需要修改交换机的硬件，FDRC 用当前数据中心的硬件就能实现，因此，适用的场景更广。

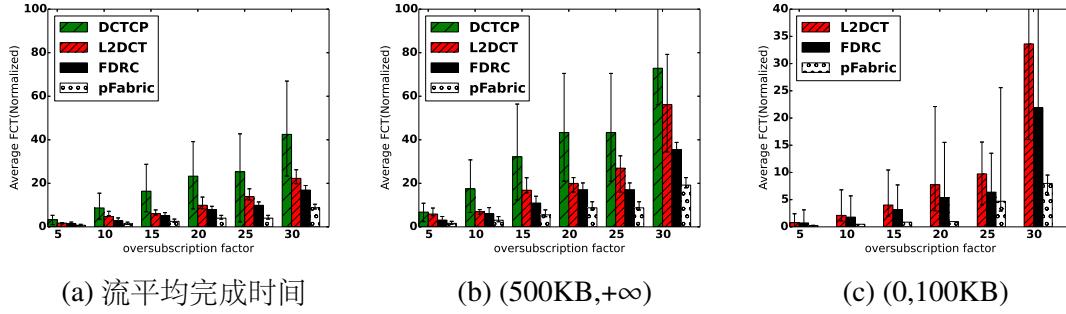


图 5.11 网络搜索流量下，FDRC, DCTCP, L²DCT, pFabric 的标准化的流平均时间对比。注意在图 (c) 中，DCTCP 没有出现其中，因为 DCTCP 的平均流完成时间太长，因此并没有画出

图 5.11 是网络搜索流量，变换链路的超额认购系数的 FDRC, DCTCP, L²DCT, pFabric，标准化的流平均时间的对比。图 5.11(a) 描述的是流平均完成时间的整体情况。发现，当超额认购系数从 5 增加到 30 时，DCTCP 的流平均完成时间从 5 增加到 70。L²DCT 的性能比 DCTCP 强，使用 L²DCT，流平均完成时间在 3 到 22 之间，L²DCT 的性能大约是 DCTCP 的 3×。FDRC 的性能优于 L²DCT，使用 FDRC 的流平均完成时间在 3 到 17 之间。FDRC 的性能比 L²DCT 提高 30%。近似最优的策略 pFabric 的性能最好，使用 pFabric 下的流平均完成时间在 2 到 10 之间。FDRC 的性能比之差 30%。图 5.11(b) 描述的是流大小在 (500KB,+∞) 时，流平均完成时间的整体情况。发现，对于这些长流，当超额认购系数从 5 增加到 30 时，DCTCP 的流平均完成时间从 5 增加到 120。使用 L²DCT，流平均完成时间在 3 到 74 之间，L²DCT 的性能比 DCTCP 提高大约 40%。FDRC 的性能优于 L²DCT，使用 FDRC 的流平均完成时间在 3 到 40 之间。FDRC 的性能比 L²DCT 提高 1×。近似最优的策略 pFabric 的性能最好，使用 pFabric 下的流平均完成时间在 2 到 20 之间。FDRC 的性能比之差 30%。图 5.11(c) 描述的是短流的流平均完成时间的整体情形。实验的结果和前面两组基本类似。

5.4.3 真实环境下的测试床测试

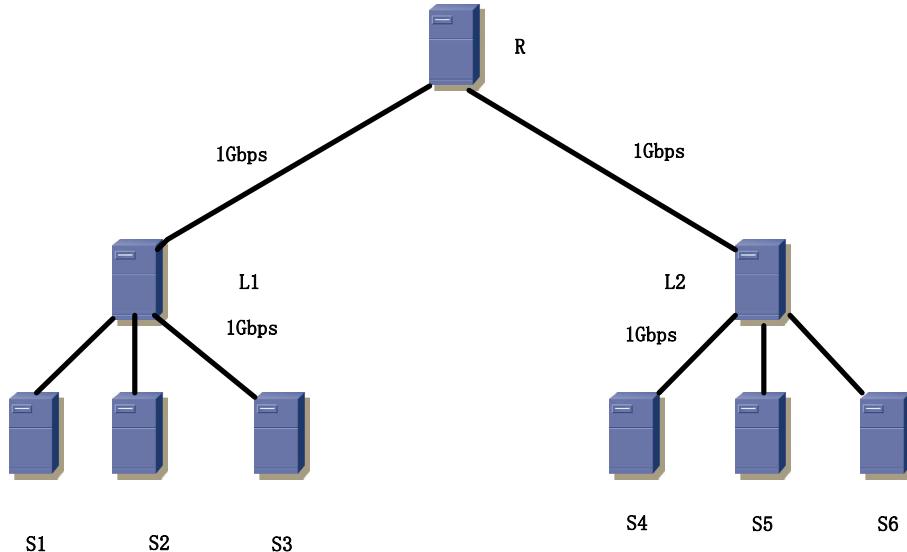


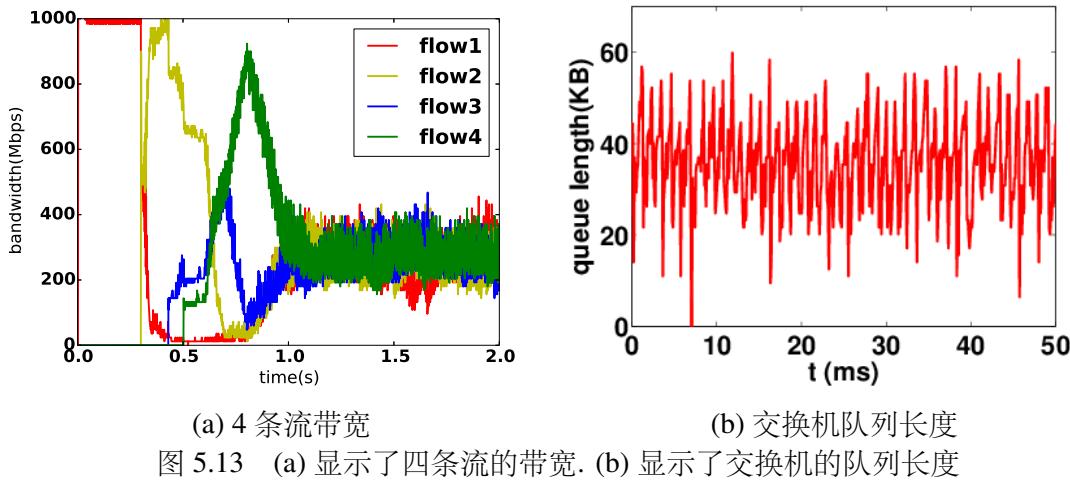
图 5.12 小型数据中心测试床

为了测试 FDRC 在真实环境下的性能，基于 DCTCP 的代码^[53]，在 Linux 内核 3.2.61 的平台下实现了 D²TCP, L²DCT, LPD。在实验中，使用 Linux 系统当作交换机，当作交换机的 Linux 系统运行的是 FreeBSD 9.1，系统上安装了 DummyNet^[55]，改进了 DummyNet 来进行数据包的标记和线速的转发。因为硬件数目的限制，建立了小型的树状的拓扑，拓扑有 3 个交换机，6 个机器，拓扑的结构如图 5.12 所示。

5.4.3.1 带宽测试

使用 FDRC，流开始有最高的优先级，随着时间的流逝，流的优先级开始降低。在本组中，测试真实环境下 FDRC 的带宽等的性能。在的测试实验中，设置 $DMAX = 0.8$, $DMIN = 0.3$ ，设置 $threshold_lax = 200\text{ms}$ ，阈值 $threshold_tight = 100\text{ms}$ ，交换机的队列 $K = 40KB$ 。启动 4 条数据流， $flow_1$ 从 S_1 启动，启动时间为 $t = 0s$, $flow_2$ 从 S_2 启动，启动时间为 $t = 0.3s$, $flow_3$ 从 S_3 启动，启动时间为 $t = 0.4s$, $flow_4$ 从 S_4 启动，启动时间为 $t = 0.5s$ ，所有流的目的地都是机器 R 。

图 5.13 展示的是流完成时间对比。从图 5.13(a) 发现，使用 FDRC 协议，流开始的平均带宽最高，随着时间的流逝，流的带宽开始下降，最后，当流的持续时间大于 200ms 时，流达到最低优先级，最终，所有的流有相同的优先级，它们共享链路带宽。图 5.13(b) 是队列，可以看到，队列长度始终保持在比较低的水平上，设置的 $K = 40KB$ ，因此交换机的队列在 40KB 波动。因此使用 FDRC 方法，能够在维持链路高带宽的基础上同时维持队列维持在比较短的水平。



(a) 显示了四条流的带宽. (b) 显示了交换机的队列长度

5.4.3.2 错失期限测试

在本小节，测试当数据中心中有延迟敏感的数据流 FDRC 的性能。在实验中，首先，每个机器 S_i 和随机的选取一台机器建立 TCP 连接。每个 TCP 连接持续 T 分钟。在 T 分钟中， S_i 将会发送短的数据流给接收端。 T 分钟过后，这台机器 S_i 会再随机选择一台机器，然后重复这个过程。在这组实验中，设置 $DMAX = 0.8$, $DMIN = 0.3$ ，设置 $threshold_lax = 200\text{ms}$ ，阈值 $threshold_tight = 100\text{ms}$ ，设置流的期限在 100ms 到 200ms 之间，并且重复这个过程 100 次，为了进行对比，同样实现了 D²TCP, LPD 和 DCTCP，变化数据包的大小，最终的实验结果如图5.14所示。

从图5.14看到，整体上，FDRC 的性能好于 DCTCP, D²TCP，但是比 LPD 要差。当数据包大小在 $50KB$ 时，FDRC 错失流的比例平均为 0.3% ，当数据包大小为 $100KB$ 时，FDRC 错失流的比例平均为 0.5% 。当数据包大小为 $150KB$ 时，FDRC 错失流的比例平均为 1% ，当数据包大小为 $200KB$ 时，FDRC 错失流的比例平均为 1.2% 。FDRC 的性能整体比 DCTCP 提升 $3\times$ ，比 D²TCP 提升 30% 。

5.4.3.3 流平均完成时间优化测试

在本小节，测试 FDRC 对流完成时间的优化。在实验中，首先，每个机器 S_i 和随机的选取一台机器建立 TCP 连接。每个 TCP 连接持续 T 分钟。在 T 分钟中， S_i 将会发送短的数据流给接收端。 T 分钟过后，这台机器 S_i 会再随机选择一台机器，然后重复此过程。在这组实验中，设置 $DMAX = 0.8$, $DMIN = 0.3$ ，设置 $threshold_lax = 200\text{ms}$ ，阈值 $threshold_tight = 100\text{ms}$ ，重复这个过程 100 次，为了进行对比，同样实现了 D²TCP, LPD 和 DCTCP，变化数据包的大小，最终的实验结果如图5.15所示。

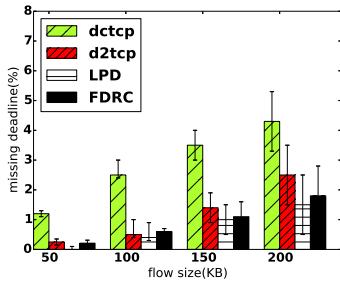


图 5.14 错失期限的流的比例

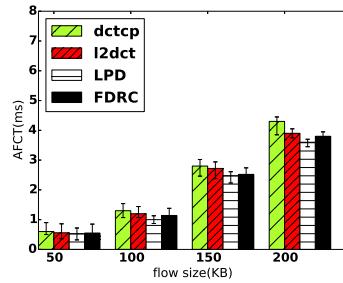


图 5.15 流平均完成时间

从图5.15看到，整体上，FDRC 的性能好于 DCTCP, L²DCT, 但是比 LPD 要差。当数据包大小在 50KB 时，FDRC 的流平均完成时间为 0.5ms；当数据包大小在 100KB 时，FDRC 的流平均完成时间为 1ms；当数据包大小在 150KB 时，FDRC 的流平均完成时间为 2.5ms；当数据包大小在 200KB 时，FDRC 的流平均完成时间为 3ms。

5.5 本章总结

在本章中，把流持续时间引入到惩罚函数的计算中。本文通过设置适当的阈值，紧急的流比背景流获得更多带宽，从而在截止期限之前完成。基于此，本文提出基于流持续时间的速率控制机制 (Flow Duration Time Rate Control, 简称 FDRC)，并且对 FDRC 建模。实验发现 FDRC 性能比 D²TCP 和 L²DCT 高 30%。尽管 FDRC 在仿真和真实环境中性能都很好，但是本章未对 FDRC 的稳定性等进行讨论。

第6章 数据中心任务级传输优化模型

6.1 本章概述

本章对数据中心任务传输模型进行介绍，首先，本章从数据中心非阻塞模型，以及传输任务等对数据中心任务级传输的模型进行介绍。随后本章介绍数据中心任务调度问题的复杂度，最后，引入任务的离线调度算法，并证明离线调度算法的近似度。

6.2 场景描述

本章首先介绍数据中心网络的非阻塞结构的定义以及非阻塞结构的特性，然后定义数据中心传输任务，以及介绍数据中心传输任务的特征。

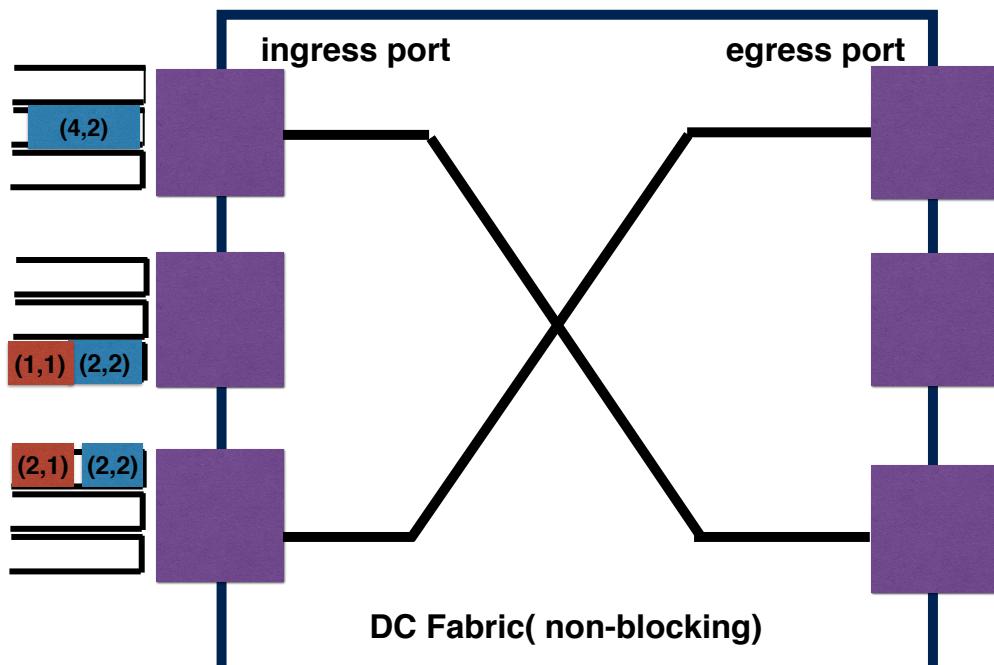


图 6.1 非阻塞的数据中心体系结构

6.2.1 网络模型

最近的一些研究方法^[32,37,39,59], 把数据中心当做一个大型交换机。如图6.1所示, 在非阻塞的网络模型中, 所有的端口有统一的转发能力, 此外, 在此网络模型中, 所有的任务竞争交换机的入端口和出端口的带宽。在本体系结构下, 数据中心的内部没有任务的竞争, 只需要在入端口和出端口对任务进行优先级调度即可。在后面的讨论中, 为了简化问题, 基于非阻塞体系结构讨论数据中心任务的调度。

6.2.2 传输任务的定义

在本部分, 定义数据中心的传输任务, 数据中心的传输任务被定义为流组 (coflow)^[37], 流组是数据流的集合, 这些流有共同的目标或者共同需要满足一定的期限等^[37,38]。考虑在 $t = 0$ 时, 数据中心同时开始传输 n 条流组 (coflow), 在非阻塞的网络体系结构下, 假设数据中心有 m 台主机, 对应在非阻塞的网络体系结构下有 m 个入端口和 m 个出端口。

下面给出数据中心流组的定义:

定义 6.1: 第 k 个流组 (coflow) $F^{(k)}$, 数学上定义为: $F^{(k)} = \{f_{i,j}^k | 1 \leq i \leq m, 1 \leq j \leq m\}$, $f_{i,j}^k$ 是每条流的长度 (标准化后), 机器 i 到机器 j 。其中 k 是 coflow 的序号。

为了进一步简化问题, 假设此“大型交换机”的入端口和出端口的带宽都是 1 (标准化后)。所以, 当只有这一条 coflow 进行传输时, 传输时间是 $f_{i,j}^k$ 。在数据中心中, 并不是每一条 coflow 都有 $m \times m$ 条并行的数据流, 如果从机器 i 到 j 没有数据传输, 那么 $f_{i,j}^k = 0$ 。下面定义 coflow 的长度和宽度,

定义 6.2: coflow k 的长度 $\text{length}(F^{(k)}) = \max_{1 \leq i \leq m, 1 \leq j \leq m} \{f_{i,j}^k\}$, 即 coflow 中传输时间最长的数据流

定义 6.3: 定义 $b_{i,j}^k$ 为第 k 条 coflow 的是否有从机器 i 到机器 j 数据传输的标记, 如果 $f_{i,j}^k \neq 0$, 那么 $b_{i,j}^k = 1$, 否则 $b_{i,j}^k = 0$ 。coflow 的宽度为: $\text{width}(F^{(k)}) = \sum_{i=1}^m \sum_{j=1}^m b_{i,j}^k$, 即 coflow k , $F^{(k)}$ 的宽度为 coflow 中最大并行的流的数量。

6.3 传输任务的离线调度模型

本部分中, 首先讨论开放并行商店问题 (concurrent open shop problem, 简称 COSP), 然后讨论数据中心的任务调度的问题复杂度, 最后, 引入一个解决数据中心传输任务离线调度问题的调度算法, 并且讨论此算法的近似度。

6.3.1 开放并行商店问题

开放并行商店模型的定义如下。考虑一组机器 $M = \{1, 2, \dots, m\}$, 每个机器可以处理一种类型的操作。定义任务集合 $N = \{1, 2, \dots, n\}$, 每个任务有 m 个子任务需要完成, 每个机器处理一个子任务。机器之间相互独立。当所有机器上的子任务完成时, 此任务完成。设 C_j 表示第 j 个任务的完成时间。设置 w_j 为第 j 个任务的权重, 问题的优化目标为 $\min \sum_{j=1}^n w_j \times C_j$

使用 w_k 表示 coflow 的权重, 权重在实际当中可以表示为 coflow $F^{(k)}$ 的重要性, C_k 为 coflow k 的完成时间, 定义如下的理想化权重流组完成时间优化问题 (Idealized Weighted Coflow Completion Time Minimization, 简称 IWCCTM) 问题:

$$\text{minimize} \quad \sum_{k=1}^n w_k C_k \quad (6-1)$$

$$\text{s.t.} \quad \forall k, j : \sum_{\forall l: C_l \leq C_k} \sum_{i=1}^m f_{i,j}^{(l)} \leq C_k \quad (6-2)$$

$$\forall k, i : \sum_{\forall l: C_l \leq C_k} \sum_{j=1}^m f_{i,j}^{(l)} \leq C_k \quad (6-3)$$

IWCCTM 目标是最小化权重流组完成时间之和, 其中 (6-2) 和 (6-3) 是对入端口和出端口有带宽的限制。对于一个 coflow $F^{(k)}$, 完成时间是 C_k , 考虑在 $F^{(k)}$ 完成之前的流组 (coflow) 的集合: $F^{(l)}: C_l \leq C_k$ 。对于任意的入端口 i (或者出端口 j) , 在这个端口上的所有流的传输时间, 至少为: $\sum_{j=1}^m f_{i,j}$ (出端口 j 上的为: $\sum_{i=1}^m f_{i,j}$), 这个值应该小于 C_k 。下面证明在只允许非抢先调度的情形下, 任务的权重完成时间之和的问题是 NP-hard 的。

6.3.2 NP-hard 证明

在本部分中, 证明, 在只允许非抢先调度的情形下, 任务的权重完成时间之和的问题是 NP-hard 的。

引理 6.1: 如果只允许非抢先调度, 那么 IWCCTM 问题和开放商店中任务的权重完成时间之和的问题是等价的, 问题的复杂度是 NP-hard。

证明 开放商店可以被描述为: 有一系列的机器 $M = \{1, \dots, m\}$, 每台机器能执行一种类型的操作, 任务集合表示为 $N = \{1, \dots, n\}$, 每个任务需要特定的 m 类的操作。每个任务 $j \in N$ 有一个权重 $w_j \in R > 0$, 并且任务 j 在机器 i 上的处理时间是

$p_{ij} \in R \geq 0$ 。当一个任务在机器上的所有操作都完成了，才能认为这个任务完成。特别的，一个任务的所有操作能够并行的处理。

对于一个 $m \times m$ 的数据中心，根据定义（6.1）可以得到一条 coflow $F^{(k)}$ ，聚合每个端口发出的数据流，得到数据中心传输子任务的定义：

$$L_l^{(k)} = \begin{cases} \sum_{i=1}^m f_{i,l}^{(k)} & 0 \leq l < m \\ \sum_{j=1}^m f_{l-m,j}^{(k)} & m \leq l < 2m \end{cases} \quad (6.4)$$

因为数据中心有 $2m$ 个端口，因此每个流组有 $2m$ 个传输子任务。考虑开放商店中有 n 个任务和 $2m$ 台转发能力相同的机器，每个任务有 $2m$ 个子任务，每个子任务需要在一台机器上执行。每个流组可以和每个任务对应，每台机器的子任务执行和每个传输子任务对应。因为已经被证明，在开放商店中最小化任务权重完成时间是 NP-hard^{[60][61][42][62]}，因此，在非阻塞网络中对流组的调度问题的复杂度也是 NP-hard。 \square

6.3.3 离线算法

对于在开放商店中解决任务权重完成时间，当前最优的解决方法是 2-近似的贪心策略^[60,63]。因为 IWCCTM 问题和开放商店任务权重完成时间有很近的关系，引入一个如算法2所示的离线非抢先调度算法。

算法2首先构造了端口集合 $\mathcal{P} = \{1, \dots, 2m\}$ ，这个集合对应着非阻塞网络体系结构的“巨型交换机”的 m 个入端口和 m 个出端口，行 1~4，计算了每个端口的负载。然后算法迭代的找出第 i 个循环下要调度的流组（coflow），循环是按照倒叙进行。在每轮迭代中，首先找出负载最重的端口 p^* ，然后找到这个端口上有最小的权重和负载比的流组（coflow），保存这个流组（coflow）的负载到 $\gamma[i]$ （行 6~7），随后更新要调度的 coflow 的集合，以及要调度的 coflow 集合的权重，和端口的负载（行 8~11），随后，进行下一轮的迭代。

可以很明显的看出，算法2产生序列 γ 的复杂度是 $O(n(m + n))$ ，其中 m 是数据中心中端数量， n 是任务的数量。

6.3.4 离线算法 $(2 - \frac{2}{n+1})$ -近似证明

本部分，证明算法2的是 2-近似，根据 Chen^[61]，IWCCM 中有如下限制：

$$\forall j \leq 2m, S \subset F : \sum_{k=1}^{|S|} L_j^{(k)} C_k \geq \frac{1}{2} \sum_{k=1}^{|S|} (L_j^{(k)})^2 + \frac{1}{2} (\sum_{k=1}^{|S|} L_j^{(k)})^2 \quad (6.5)$$

Algorithm 2: 解决 IWCCM 的 2 近似算法

Input: Coflow 集合 $\mathcal{F} = \{F^{(k)}\}$, 权重集合 $\mathcal{W} = \{\overline{w}_k = w_k\}, 1 \leq k \leq n$;

Output: a permutation γ of $\{1, \dots, n\}$;

- 1 $\mathcal{P} \leftarrow \{1, \dots, 2m\}, \mathcal{R} \leftarrow \{1, \dots, n\};$
 - 2 $L_i^{(k)} = \sum_{j=1}^m f_{i,j}^{(k)}$ for $1 \leq k \leq n$ and $i \leq m$;
 - 3 $L_{j+m}^{(k)} = \sum_{i=1}^m f_{i,j}^{(k)}$ for $1 \leq k \leq n$ and $j \leq m$;
 - 4 $L_p = \sum_{1 \leq k \leq n} L_p^{(k)}$ for each $p \in \mathcal{P}$;
 - 5 **for** i from n to 1 **do**
 - 6 $p^* = \arg \max_{p \in \mathcal{P}} L_p$;
 - 7 $\gamma[i] = r^* = \arg \min_{r \in \mathcal{R}} \overline{w}_r / L_{p^*}^{(r)}$;
 - 8 $\theta = w_{r^*} / L_{p^*}^{(r^*)}$;
 - 9 $\mathcal{R} = \mathcal{R} \setminus \{r^*\}$;
 - 10 $\overline{w}_r = \overline{w}_r - \theta \times L_{p^*}^{(r)}$ for all $r \in \mathcal{R}$;
 - 11 $L_p = L_p - L_{p^*}^{(r^*)}$ for all $p \in \mathcal{P}$;
 - 12 **return** γ ;
-

其中 F 是要调度的 coflow 的集合, $L_j^{(k)}$ 是 coflow k 在端口 j 的传输时间, $|S|$ 是集合 S 的长度, 因此, IWCCM 可以表述为:

$$\text{minimize } \sum_{k=1}^n w_k C_k \quad (6-6)$$

$$\text{s.t. } \forall j \leq 2m, S \subset F : \sum_{k=1}^{|S|} L_j^{(k)} C_k \geq \frac{1}{2} \sum_{k=1}^{|S|} (L_j^{(k)})^2 + \frac{1}{2} (\sum_{k=1}^{|S|} L_j^{(k)})^2 \quad (6-7)$$

基于此, 得到 IWCCM 的对偶函数为:

$$\text{maximize } \sum_{j=1}^{2m} \sum_{S \subset F} \left[\frac{1}{2} \sum_{k=1}^{|S|} (L_j^{(k)})^2 + \frac{1}{2} (\sum_{k=1}^{|S|} L_j^{(k)})^2 \right] y_j^S \quad (6-8)$$

$$\text{s.t. } \forall k \leq n : \sum_{j=1}^{2m} L_j^{(k)} \sum_{S \subset F: k=1} y_j^S = w_k \quad (6-9)$$

$$\forall y_j^S \geq 0 \quad (6-10)$$

为了证明算法2的是 2-近似, 引入以下引理:

引理 6.2: 对任意的 $j \leq 2m$ 和 $S \subset F$, 有 $(\sum_{k=1}^{|S|} L_j^{(k)})^2 \leq (2 - \frac{2}{n+1}) \left[\frac{1}{2} \sum_{k=1}^{|S|} (L_j^{(k)})^2 + \frac{1}{2} (\sum_{k=1}^{|S|} L_j^{(k)})^2 \right]$

基于此, 有如下定理:

定理 6.1: 算法2的近似度是 $(2 - \frac{2}{n+1})$, 其中 n 是并行的 coflow 的数目

证明 使用 $p(i)$ 表示在第 6 行中第 i 个循环中选出的端口, 使用 $\theta(i)$ 表示在第 8 行中第 i 个循环计算的 θ 值, 使用 $\overline{w_r(i)}$ 表示第 10 行中第 i 个循环中调整的权重, 使用 $\gamma(i)$ 表示第 i 循环起始时没有调度的 coflow 集合。有 $\gamma(i) = \{r^*(1), r^*(2) \dots r^*(i)\}$, 其中 $r^*(i)$ 表示 coflow 的 id, 定义如下的对偶解, 对于任意的 $j \leq 2m$ 和 $S \subset F$

$$y_j^S = \begin{cases} \theta(i) & j=p(i), S=\gamma(i), i=1,2,3\dots n \\ 0 & \text{其它情况} \end{cases} \quad (6-11)$$

(6-11) 是 (6-8) 的对偶解, 是因为:

$$\begin{aligned} \sum_{j=1}^{2m} L_j^{r^*(i)} \sum_{S \subset F} y_j^S &\stackrel{(a)}{=} \sum_{l=i}^n L_{p(l)}^{r^*(i)} y_{r^*(l)}^S \\ &\stackrel{(b)}{=} \sum_{l=i}^n L_{p(l)}^{r^*(i)} \theta(l) \\ &\stackrel{(c)}{=} w_{r^*(i)} - \overline{w_{r^*(i)}} \\ &\stackrel{(d)}{=} w_{r^*(i)} \end{aligned} \quad (6-12)$$

(a) 成立是因为算法 1 是从 n 到 1 开始循环的, 因此第 i 个循环的计算结果可以通过 $i+1$ 个循环到 n 个循环推算得到。(b) 成立是因为有解 (6-11) 成立, (c) 成立是因为在第 i 个循环, $r \in \gamma(i)$ 有

$$w_r^{(i)} = w_r - \sum_{l=k}^n L_{p(l)}^r \theta(l) \quad (6-13)$$

(d) 成立是因为对于任意的 $k=1,2..n$, 有 $\overline{w_{r^*(l)}} = 0$ 。通过 (6-12), 可以得知, 解 (6-11) 满足 (6-9)。此外, 从算法2输入得知, $\overline{w_{r^*(l)}}$ 初始化值为权重初始值, 算法2第 7 行和第 10 行知, 每个循环选出最小的权重负载比, 因此, 第 10 行中, $\theta \geq 0$, 因此公式 (6-10) 成立。下面证明算法2的近似度是 $(2 - \frac{2}{n+1})$ 。注意, 算法2调度的 coflow 满

足: $C_{\gamma[1]} \leq C_{\gamma[2]} \leq \dots C_{\gamma[n]}$, 根据步骤 6, 7, 对所有的 $i=1,2,\dots,n$, 有 $C_{\gamma[i]} = \sum_{j=1}^i L_{p^*(i)}^{r^*(j)}$ 。令 $(C_k^{CP})_{k \in \mathcal{F}}$ 是最优解, $(C_k^*)_{k \in \mathcal{F}}$ 是最优解向量。调度的最优化解为:

$$\begin{aligned}
 & \sum_{k=1}^n w_k C_k \\
 &= \sum_{k=1}^n \left(\sum_{j=1}^{2m} L_j^{(k)} \sum_{S \subset F} y_j^S \right) C_k \\
 &= \sum_{j=1}^{2m} \sum_{S \subset F} y_j^S \sum_{k \in S} L_j^{(k)} C_k \\
 &= \sum_{l=1}^n y_{p^*(l)}^{\gamma(l)} \sum_{g=1}^l L_{p^*(l)}^g C_g \\
 &= \sum_{l=1}^n y_{p^*(l)}^{\gamma(l)} \sum_{h=1}^l L_{p^*(l)}^{r^*(h)} C_{r^*(h)} \\
 &\stackrel{(e)}{\leq} \sum_{l=1}^n y_{p^*(l)}^{\gamma(l)} C_{r^*(l)} \sum_{h=1}^l L_{p^*(l)}^{r^*(h)} \\
 &\stackrel{(f)}{=} \sum_{l=1}^n y_{p^*(l)}^{\gamma(l)} \left(\sum_{h=1}^l L_{p^*(l)}^{r^*(h)} \right)^2 \\
 &\stackrel{(g)}{\leq} \left(2 - \frac{2}{n+1} \right) \sum_{l=1}^n y_{p^*(l)}^{\gamma(l)} \left\{ \frac{1}{2} \sum_{h=1}^l \left[L_{p^*(l)}^{r^*(h)} \right]^2 + \frac{1}{2} \left[\sum_{h=1}^l L_{p^*(l)}^{r^*(h)} \right]^2 \right\} \\
 &\stackrel{(h)}{\leq} \left(2 - \frac{2}{n+1} \right) \sum_{k=1}^n w_k C_k^{CP} \\
 &\leq \left(2 - \frac{2}{n+1} \right) \sum_{k=1}^n w_k C_k^* \tag{6-14}
 \end{aligned}$$

(e) 成立因为对于 $h=1,2,\dots,1$, 有 $C_{r^*(h)} \leq C_{r^*(l)}$, (f) 成立, 因为 $C_{r^*(l)} = \sum_{h=1}^l L_{p^*(l)}^{r^*(h)}$, (g) 成立因为有引理6.2成立, (h) 成立因为 y 是 (6-8)~(6-10) 的解。因此算法2的近似度是 $\left(2 - \frac{2}{n+1} \right)$, 其中 n 是并行的流组的数目 \square

6.4 本章小结

在本章中, 介绍了数据中心任务级传输优化模型, 首先介绍了数据中心非阻塞模型, 随后证明了数据中心流组 (coflow) 调度问题的复杂度, 最后提出了一个 2-近似算法并证明算法的近似度。

第7章 基于流信息的任务级传输优化方案

随着大数据应用的部署，越来越多的数据存储在大数据存储系统中。大数据存储系统希望给用户提供安全，稳定，高效的文件存取服务。纠删码存储系统已被诸如 Google 和 Facebook 等公司广泛使用，因为它使用纠错码来保证文件系统的可靠性。在纠删码存储系统中，使用 (n, k) MDS 纠删码用于将文件分成 n 个块。当用户想要访问文件时，文件系统可以选取 n 个块中的任何 k 个子集来重建文件。在这种情况下，如何从 n 个数据块中选出 k 个数据集，如何高效的传输这 k 个数据块，成为一个重要的问题。本章提出基于流信息的任务级传输优化方法来最小化文件平均访问时间（File Access Time，简称 FAT）。为了实现这一点，本章同时结合了最小负载优先的启发式算法做为从 N 个数据块中选取 K 个数据块的方法。在此基础上，本章设计并实现了 D-Target，一个集中式调度器，对文件系统的源选择以及文件系统的任务传输进行整体优化。最后，通过仿真实验来验证 D-Target 的性能，结果表明，对于 AT&T 的文件存储实验数据，D-Target 在 FAT 性能方面分别比 TCP，Aalo，Barrat 和 pFabric 分别提高了 $2.5\times$, $1.7\times$, $1.8\times$ 。

7.1 概述

随着大数据应用的发展，越来越多的数据被存储于在线存储系统中。此外，企业正在依靠大数据分析技术来实现商业的智能化，并正在将其传统 IT 基础架构迁移到云中。在这种趋势下，亚马逊的云端硬盘，苹果的 iCloud，DropBox，Google Drive，微软的 SkyDrive，AT&T Locker 等云服务，分布式的部署在云平台上，并且采用各种冗余备份的方式，以提高云存储系统的稳定性。

纠删码已被 Google 和 Facebook 等公司广泛使用^[64,65]，因为它提供了空间最优的编码方式来防止数据丢失。在纠删码存储系统中，使用 (n, k) MDS 纠删码用于将文件分成 n 个块。当用户获取文件时，需要 n 个块中的任意 k 个子集来重建它。如图7.1所示，文件的数据块存储在云服务器上，客户端收到来自客户的请求，运行在客户端上的调度程序为每个请求选择数据源。然后，客户端接收块，重建文件并将文件发送给用户。用户最关心的是文件获取的速度（File Access Time，简称 FAT）。

可以看到两个过程影响分布式纠删码存储系统的文件访问速度。首先，效率低下源选择会导致数据块的选择冲突。分布式纠删码存储系统总是使用随机源选择请求，随机源选择可以均分请求，但是，随机源选择会使一些节点的负载很重，

因为节点接入到网络中的带宽是有限的，使用随机选取数据源的方法会导致很多请求选取同一台机器作为数据块传输的源端，从而同一台机器上发生冲突，从而影响任务的传输效率。为防止这种情况发生，应该选择负载较小的服务器来减少冲突。其次，TCP 不是分布式存储系统的好选择。TCP 是公平的传输方式，但是在纠删码存储系统中，只有客户端接收到所有的数据块才能开始重建文件。对于 TCP 传输，不同的块可能获得不同的带宽，最后完成传输的数据块可能会传输的很慢，这对带宽造成浪费，提前完成的数据传输需要等待后面的传输完成。如果把早传输完成的数据块的带宽分配给其余的请求任务，那么就能提高链路资源的利用率。

本章介绍的方法，试图最小化分布式纠删码存储系统的文件平均访问时间 (FAT)。将源选择和数据块传输联合起来进行优化。首先，提出理想情况下的文件访问时间最小化 (Idealized File Access Time Minimization, 简称 IFATM) 问题。然后忽略源选择，假设数据源是固定的，提出简单理想化的文件访问时间最小化 (Simple Idealized File Access Time Minimization, 简称 SIFATM) 问题。然后，证明即使简单的理想化的问题 SIFATM 也是 NP-hard。在此之后，提出启发式的“最小负载优先”策略，作为源选择的策略。对于数据块的传输，使用 2 阶在线近似的方法来决定数据块的传输次序，并使用最小化期望分配带宽 (Minimum-Allocation-

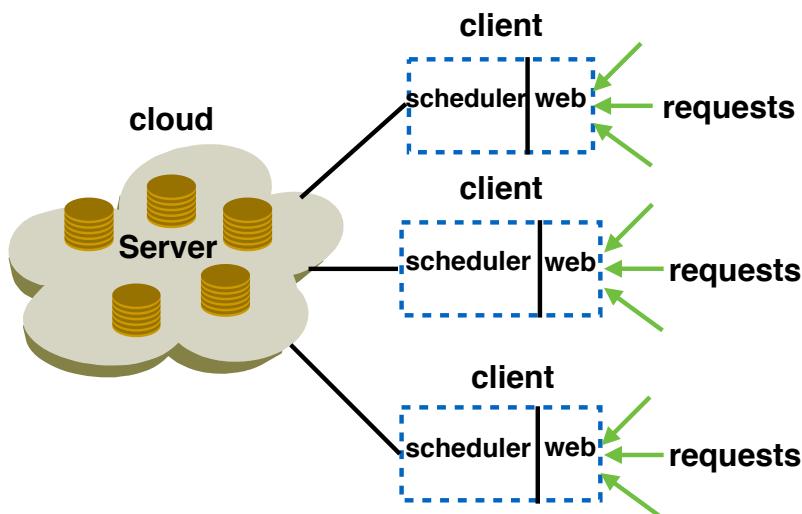


图 7.1 纠错存储系统的结构图。数据块被存储在云中的机器中，客户端接收用户的请求，然后选择相应的节点来进行数据的组装

for-Desired-Duration, 简称 MADD) 来计算要分配的带宽。然后设计了一个调度器 D-Target, 使用它来计算数据块发送需要的带宽。最后用 AT&T 的数据集来测试 D-Target 的性能。结果表明, D-Target 分别比 TCP, Aalo^[39], Barrat^[36] 和 pFabric^[32] 性能分别提高了 2.5×, 1.7×, 1.8×, 3.6×。在本章节中做出的贡献是:

- (1) 在分布式纠删码存储系统中, 提出将源选择和数据块的传输结合起来进行优化。
- (2) 提出理想情况下的文件访问时间最小化 (Idealized File Access Time Minimization, 简称 IFATM) 问题, 研究其复杂度, 并导出非近似优化的非抢先式调度算法。
- (3) 设计 D-Target, 一个调度器来选择有效的信号源并为中继线分配带宽。实验结果显示, D-Target 分别比 TCP, Barrat^[36], Aalo^[39], pFabric^[32] 提高了 2.5×, 1.7×, 1.8×, 3.6×。

7.2 研究动机和相关工作

7.2.1 研究动机

纠删编码已经在分布式存储系统中被广泛的使用。在纠错码存储系统中, 文件被编码成数据块, 数据块被存储在不同的机器上。在访问文件时, 应该从存储块节点的机器集合中选择机器来重建文件。从用户角度看, 用户关心的是请求获

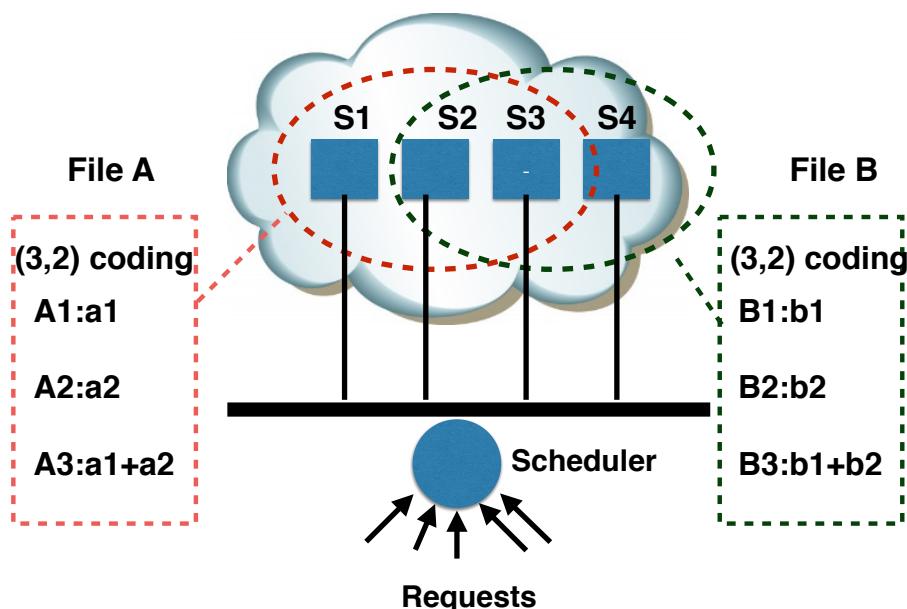


图 7.2 一个纠删码存储系统, 存储了两个文件, 使用 (3, 2) MDS 进行编码

得相应的时间。请求响应的时间越短，用户获得的体验越好，从而系统获得的用户满意度越高。但是，云中节点之间，节点接入网络的带宽有限，这会增加用获取文件的时间长度，这会影响用户的体验。在本节中，展示了纠删编码存储系统的源选择和任务级传输优化的必要性。

在纠删码存储系统中，文件使用（MDS）码编成块。文件使用（n, k）MDS编码进行编码时，一个文件被编码并存储在 n 个存储节点中。n 个节点中任何 k 个节点都能通过编码运算重新组装成文件。如图7.2所示，考虑两个文件 A 和 B，它们都被编码并存储在 $n = 3$ 个机器上。在从 $\{S_1, S_2, S_3\}$ 中选择的 2 个不同的节点成功处理之后，可以完成检索文件 A 的请求。文件 B 类似，可以从 $\{S_2, S_3, S_4\}$ 中选择。为了简化问题，假设所有链接的容量为 1，文件 A 和文件 B 的块大小也为 1。首先，考虑在 $t = 0$ 时同时到达的两个请求 R_A 和 R_B 分别取文件 A 和文件 B。对于 R_A ，文件 A 具有 3 个源选择选项： (S_1, S_2) , (S_1, S_3) , (S_2, S_3) ，而文件 B 有 (S_2, S_3) , (S_2, S_4) , (S_3, S_4) 三个源可供选择。如果调度器使用随机源选择，则有一种常见的情况是 R_A 选择 (S_2, S_3) , R_B 选择 (S_3, S_4) 。如果传输策略是 TCP，那么 R_A 的文件访问时间为 $t_A = 2$ ，对于 R_B ，则为 $t_B = 2$ ，那么平均文件访问时间（Average File Access Time，简称 AFAT=2。但是，如果 R_A 选择 (S_1, S_2) 和 R_B 选择 (S_3, S_4) ，AFAT=1。从简单的例子可以看出，对于纠删码存储系统，高效的源选择可以减少 AFAT，从而用户体验会更好。

而且，对于分布式存储系统，公平传输的方法 TCP 并不能达到好的效果。试想一下简单的情况，文件 A 有两个请求， R_{A_1} 到达时间为 $t = 0$, R_{A_2} 到达时间为 $t = 0.1$ 。 R_{A_1} 选择 (S_1, S_2) 作为源， R_{A_2} 选择 (S_2, S_3) 作为源。对于 TCP 传输， R_{A_1} 的文件访问时间 $FAT=1.9$, R_{A_2} 的文件访问时间 $FAT=1.9$ ，所以 $AFAT= (1.9 + 2.0) / 2 = 1.95$ 。但是，如果让 R_{A_1} 具有比 R_{A_2} 更高的优先级。那么 R_{A_1} 的文件访问时间 $FAT= 1$ 和 R_{A_2} 的文件访问时间 $FAT = 2$ ，则 $AFAT= (1 + 2) / 2 = 1.5$ 。可以看到，在分布式存储系统中，结合任务级优化可以大大减少 AFAT。

从这两个简单的例子中，可以看到源选择和文件传输优化在减少文件访问时间方面起着重要的作用。本章节将这两个问题结合起来进行优化。

7.2.2 相关工作

根据调查发现，2011 年仅用于照片存储的空间已经超过 20 PB，而且每周增加 60 TB^[66,67]。磁盘上进行着大量的海量存储，每天都可能出现大量故障。为了弥补数据丢失，保证存储在云中的数据的完整性，需要对数据进行多分备份，并且将副本存储在多个磁盘。这样做的优势是可以容忍数据的丢失，因为只要有一个备

份可用就能迅速的得到需要的数据^[67]。但是，仅仅存储数据的备份，会影响磁盘空间的使用效率。使用纠删编码将数据编码成块，把数据块分布式存储在集群中，当用户请求数据时，可以选择其中集群的子集数据块，通过解码算法，得到相应的数据。这种方法可以弥补数据丢失，从而保证存储在云中的数据的完整性。

尽管纠删编码存储系统可以保证数据的完整性，但是由于读写数据，系统需要对数据进行编码或解码，由于 CPU 的限制，经常导致数据访问延迟高，因为网络访问吞吐量低，^[68-70] 主要关注高效编码和解码技术。通过高效的编码技术，CPU 的开销将大大降低。然而，尽管纠删编码将数据存储为多个编码块，但是当访问文件时，系统必须从数据块的存储集合中选择一定数目的子块来进行文件的重组。文件数据块的传输过程会给网络增加很大的负担。事实上，延迟优化已经引起越来越多的关注，甚至谷歌和亚马逊已经公布，每增加 500 毫秒可能导致 1.2% 的用户损失^[71]。优化传输延迟非常重要，因为这可以改善用户体验，从而增加收入。现在还有很多方法可以减少应用程序在数据中心的传输延迟。根据优化的粒度，可以将其划分为流粒度优化和任务粒度优化两种。

DCTCP^[16], D²TCP^[23], L²DCT^[24], PDQ^[28], pFabric^[32], LPD^[57], D³^[27] 是流级别的优化方法。DCTCP 是公平的共享方法，它试图保持交换机的队列短，以减少队列延迟。然而，在数据中心，应用对带宽有不同的需求，公平分享不是一个好的选择^[57]。D²TCP, LPD, D³ 是根据截止期限和网络拥塞情况进行带宽分配方法。他们设定了明确的截止期限，以适应不同的紧急情况，从而使期限紧的数据流获得比期限宽的数据流有更多的带宽。结果是，更多的应用程序获得了自己满意的带宽。L²DCT, PDQ, pFabric 尽量减少平均流量完成时间。他们认为短流量始终是需要尽快完成传输的。他们扼制大流的带宽，并将更多的带宽分配给短流。从而，流的平均完成时间大量减少。尽管流级别的优化方法在延迟优化方面比 TCP 强，但是在数据中心网络中，分布式应用总是有并行流，只有流级优化是不够的。Barrat^[36], Varys^[37], Aalo^[39], sunflow^[59] 等任务级别的调度方法将应用程序的流程视为一个整体。Barrat^[36] 以 FIFO 的顺序安排任务，通过动态地改变网络中流的多路复用的级别来避免头部阻塞。Varys^[37] 将应用程序的并行流作为 coflow，它使用 SEBF 调度 coflows 和 MADD 执行速率控制，结果平均任务完成时间将减少。Aalo^[39], sunflow^[59] 和 CODA^[40] 事先不需要知道 coflow 的物理信息（coflow 长度，宽度等）。在纠删码存储系统中，文件访问会产生并行流，实际中流级别优化是不够的，应考虑任务级别的调度方法来提高传输效率。

7.3 网络模型和分析

在本节中，假设数据中心是无阻塞的结构，数据中心的体系结构如图6.1所示，在这个体系结构上，提出理想化文件访问时间最小化问题（Idealized File Access Time Minimization，简称 IFATM）。最后，简化这个问题，证明了简单的 IFATM 问题是 NP-hard。

7.3.1 问题分析

考虑一个由 n 个存储节点组成的非阻塞数据中心，用 $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$ 表示这些节点的组合。用 $\mathcal{F} = \{f_1, f_2, f_3, \dots, f_r\}$ 表示存储在机器集合中的 r 个文件集合。对于每个文件 f_i ，将它分成 k_i 大小的块，然后使用 (n_i, k_i) MDS 纠错码为 f_i 生成相同大小的 n_i 个不同的块。然后把不同的数据块存储在 n_i 机器上。 (n_i, k_i) MDS 纠错码的使用允许文件从 k_i -out-of- n_i 个块的任何子集重建的，而它也引入了一个冗余因子 n_i/k_i 。 m 个客户的集合，用 $\mathcal{C} = \{c_1, c_2, \dots, c_m\}$ 表示。假设所有 L 个请求在时间 0 到达，并且用 $T_{f^k}^{(k)}$ 表示第 k 个请求获取文件 f^k ($f^k \in \mathcal{F}$)。文件 f^k ($f^k \in \mathcal{F}$)。第 k 个请求 $T_{f^k}^{(k)}$ 由平行的子任务组成，每个子任务是一个流程 f^k 从服务器到客户端。 $T_{f^k}^{(k)} = \{t_{i,j}^{(k,f^k)} | 1 \leq i \leq n, 1 \leq j \leq m\}$ ，其中 $t_{i,j}^{(k,f^k)}$ 表示 $x_{i,j}^{(k,f^k)} \in \{0, 1\}$ 一个大小为 $t_{i,j}^{(k,f^k)}$ 的子任务。 $x_{i,j}^{(k,f^k)} \in \{0, 1\}$ ，用 1 表示存在从服务器 i 到客户端 j 的流。0 表示从服务器 i 到客户端 j 没有流。由于非阻塞模型的入口和出口端口有单位容量，所以传输时间对于子任务 $t_{i,j}^{(k,f^k)}$ 是 $t_{i,j}^{(k,f^k)}$ 。理想情况下的非抢先调度文件访问时间最小化问题 – Idealized File Access Time Minimization (IFATM) 可以定义为：

$$\underset{\text{minimize}}{\sum_{g=1}^L C_g} \quad (7-1)$$

$$\text{s.t.} \quad \forall g, j \sum_{\forall l: C_l \leq C_g} \sum_{i=1}^n t_{i,j}^{(l,f^l)} * x_{i,j}^{(l,f^l)} \leq C_g \quad (7-2)$$

$$\forall g, i \sum_{\forall l: C_l \leq C_g} \sum_{j=1}^m t_{i,j}^{(l,f^l)} * x_{i,j}^{(l,f^l)} \leq C_g \quad (7-3)$$

$$\forall l, j \sum_{i=1}^n x_{i,j}^{(l,f^l)} = k_{f^l} \quad (7-4)$$

目标是最小化文件平均访问时间。约束条件 (7-2) 和 (7-3) 主要是端口转发能力的限制。的优化目标是最小化文件平均访问时间。对于一个请求， $T_{f^k}^{(k)}$ ，请求的完成时刻是 C_k ，考虑到在这个时刻之前完成的文件传输的集合， $T_{f^l}^{(l)}: C_l \leq$

C_k 。对于任何一个入端口（或者出端口），这个端口上的文件平均访问时间至少是 $\sum_{\forall l: C_l \leq C_g} \sum_{i=1}^n t_{i,j}^{(l,f^l)} * x_{i,j}^{(l,f^l)}$ （或者 $\sum_{\forall l: C_l \leq C_g} \sum_{j=1}^m t_{i,j}^{(l,f^l)} * x_{i,j}^{(l,f^l)}$ ），不应该比 C_k 要大。 $(7-4)$ 是源选择的限制，意味着对于文件 f^l ，调度器要从 k_{f^l} 个机器中选择合适的机器集合。

7.3.2 NP-hard 证明

在这一部分，证明非抢先调度文件访问时间最小化问题（IFATM）问题是 NP-hard 的。为了证明这一点，首先考虑一个简单的情况，假设在一组请求中，有 $n_i = k_i$ 和 n_i 等于服务器总数。这意味着每个服务器都存储每个文件编码的一个数据块。进一步假设所有的请求同时到达。结果，简单非抢先调度文件访问时间最小化问题（Simple idealized File Access Time Minimization，简称 SIFATM）可以被定义为：

$$\underset{g=1}{\text{minimize}} \quad \sum_{g=1}^L C_g \quad (7-5)$$

$$\text{s.t.} \quad \forall g, j \quad \sum_{\forall l: C_l \leq C_g} \sum_{i=1}^n t_{i,j}^{(l,f^l)} \leq C_g \quad (7-6)$$

$$\forall g, i \quad \sum_{\forall l: C_l \leq C_g} \sum_{j=1}^m t_{i,j}^{(l,f^l)} \leq C_g \quad (7-7)$$

引理 7.1：简单非抢先调度文件访问时间最小化问题（SIFATM）是 NP-hard 问题。

证明 把章节6.3.1公式(6-1)~(7-3)中权重设置为1，那么 SIFATM 和 IWCCTM 问题相同，根据证明6.1，IWCCTM 的复杂度是 NP-hard，因此，SIFATM 也是 NP-hard。□

7.4 算法和分析

在这一节中，首先介绍一种2近似算法来解决 SIFATM 问题。然后，提出一个启发式来源来优化 IFATM 问题。最后，将算法扩展到一个可以在实践中使用的在线算法。

7.4.1 SIFATM 离线算法

SIFATM 问题相当于最小化并行开放的商店中任务完成时间问题。根据最小化并行开放的商店的完成时间与 SIFATM 问题的关系，改进最小化并行开放的商店中2-近似优化算法来解决 SIFATM 问题，改进算法如3所示。

Algorithm 3: SIFATM 的 2 近似算法

Input: 请求集合 \mathcal{T} ; 第 k 个请求, 从服务器 i 到客户端 j 发送的数据块大

小 $t_{i,j}^{(k,f^k)}$, 其中, $1 \leq i \leq n, 1 \leq j \leq m$

Output: γ

```

1  $\gamma : \{1, 2, \dots, l\} \leftarrow \mathcal{T};$ 
2  $UT \leftarrow \{1, 2, 3, \dots, l\};$ 
3  $P \leftarrow \{1, 2, 3, \dots, m+n\};$ 
4  $W\{1, 2, \dots, l\} \leftarrow \{1, 1, \dots, 1\};$ 
5  $L_i^{(k)} = \sum_{j=1}^m t_{i,j}^{(k,f^k)}$  for all  $k \leq l$  and  $i \leq n$ ;
6  $L_{j+n}^{(k)} = \sum_{i=1}^n t_{i,j}^{(k,f^k)}$  for all  $k \leq l$  and  $j \leq m$ ;
7  $L_i = \sum_{k \leq l} L_i^{(k)}$  for all  $i \in P$ ;
8 for  $i \in \{l, l-1, l-2, \dots, 1\}$  do
9    $u = \arg \max_{k \in P} L_k;$ 
10   $\gamma[i] = \arg \min_{F \in UT} W[F]/L_u^{(F)};$ 
11   $\theta = W[\gamma[i]]/L_u^{(F)}$ ;
12   $W[j] = W[j] - \theta * L_u^{(F)}$  for all  $j \in UT$ ;
13   $L_j = L_j - L_j^{(F)}$  for all  $j \in P$ ;
14   $UT = UT \setminus \{\gamma[i]\};$ 
15 return  $\gamma$ ;

```

算法3将 n 个请求的列表 \mathcal{T} 作为输入。它输出 γ , 这是所有请求的一个排列, 这个排列代表请求的调度顺序。算法首先组成一个端口列表 $P = \{1, \dots, 2m\}$, 并计算每个端口的总负载 (行 5~6)。行 8~14 是每一轮迭代的选择被调度的请求。在每次迭代中, 首先找到负载最大的端口, 然后选择具有最小权重负载比的请求, 并将其索引保存到 $\gamma[i]$ (第 10 行) 中。随后进行权重和负载的更新 (12 行和 13 行), 然后进行下一次迭代。根据定理 (6.1), 可以证明对于问题 SIFATM, 算法3是 2 近似算法。

7.4.2 从离线到在线

事实上, 算法3是一个理想的情况, 因为它假定所有请求都同时到达, 并且文件每个服务器为存储每个文件的一个数据块。事实上, 在现实世界中, 请求可以在任何时候到达, 并且不是每个机器上都存储了要请求的文件的数据块, 文件只存储机器的一个子集。在这种情况下, 算法3不是一个好的选择。

在实际中，对文件的请求会随时到达，并且端口的负载是在随时变化的。平均的，在数据中心中，因为任务的分配是负载均衡的^[36,41,72]。因此，调度器不考虑端口负载的差异性(近似1)。此外，对于分布式的纠删码存储系统，每个客户端有每个文件压缩的数据块的数据块大小(事实1)，因此，只需要在服务器端计算端口的负载即可。此外，对于纠删码存储系统，每个数据块的大小相同，因此，可以通过数据压缩比来计算要传输的数据块的大小(事实2)。定义文件 f_b 的压缩比为 $\alpha_{f_b} = \frac{\text{chunk size of } f_b}{\text{file size of } f_b}$ ，算法4是一个在线调度策略。

Algorithm 4: 在线调度算法

Input: 活跃的请求集合 \mathcal{T} , 为文件已经选好的集合 $\theta_{f^k}^{(k)} = \{x_{1,1}^{(k,f^k)}, x_{1,2}^{(k,f^k)} \dots x_{i,j}^{(k,f^k)} \dots\}, \forall T_{f^k}^{(k)} \in \mathcal{T}, \forall x_{i,j}^{(k,f^k)} \in \{0, 1\}, i \leq n, j \leq m$, 文件压缩比 $\alpha = \{\alpha_{f_1}, \alpha_{f_2} \dots \alpha_{f_r}\}$, 文件大小集合 $\beta = \{\beta_{f_1}, \beta_{f_2} \dots \beta_{f_r}\}$, 新到达的请求 T_{fc}

Output: γ

- 1 $\theta_{fc} = SourceSelection(\mathcal{T}, \theta, \alpha, \beta, T_{fc});$
 - 2 $\mathcal{T} = \mathcal{T} \cup \{T_{fc}\};$
 - 3 $L_i^{(b)} = \sum_{j=1}^m \alpha[f^b] * \beta[f^b] * x_{i,j}^{(b,f^b)}$ for $\forall b \in \mathcal{T}, i \leq n;$
 - 4 $L_{j+n}^{(b)} = \sum_{i=1}^n \alpha[f^b] * \beta[f^b] * x_{i,j}^{(b,f^b)}$ for $\forall b \in \mathcal{T}, j \leq m;$
 - 5 $l^{(b)} = \max_{1 \leq i \leq n+m} L_i^{(b)}$ for $\forall b \in \mathcal{T};$
 - 6 $\pi^{(b)} = 1/l^{(b)}$ for $\forall b \in \mathcal{T};$
 - 7 对 $[\pi^{(1)}, \pi^{(2)}, \pi^{(3)} \dots]$ 非降序排列，然后把排序后的结果给 γ ;
 - 8 **return** $\gamma;$
-

当一个新的文件请求到达时，算法4被调用。算法4的输入是 $\mathcal{T}, \theta_{f^k}^{(k)}, \alpha, \beta, T_{fc}$ ，其中 \mathcal{T} 是活跃的文件请求集合。 $\theta_{f^k}^{(k)}$ 包含源节点集合和 $T_{f^k}^{(k)}$ 的目标客户端。 α 是存储每个系统中文件大小的集合， β 是每个文件的压缩比， T_{fc} 是新到达的请求。算法4的第1行是为这个新到达的请求选择合适的选源节点，第3~5行为每个在集合 \mathcal{T} 中的请求计算负载。第3行计算的是服务器端的负载，第4行计算的是客户端的负载。特别的，使用 $\alpha[f^b] * \beta[f^b] * x_{i,j}^{(b,f^b)}$ 来计算文件 f^b 的块大小。第5行计算每个请求的负载，请求的完成时间是由最后一条流的完成时间决定的。第6行计算请求 $T_{fc}^{(c)}$ 的完成时间，同样的请求的完成时间是由最后一条流的完成时间决定的。第7行是对请求 $\pi^{(1)}, \pi^{(2)}, \pi^{(3)} \dots$ 进行非降序排序，然后把排序的结果反馈给 γ 。第8行，算法将计算的结果返回。

7.4.3 源选择问题

Algorithm 5: 最小负载优先策略

Input: 活跃的请求集合 \mathcal{T} , 为文件已经选好的集合 $\theta_{f^k}^{(k)} = \{x_{1,1}^{(k,f^k)}, x_{1,2}^{(k,f^k)} \dots x_{i,j}^{(k,f^k)} \dots\}, \forall T_{f^k}^{(k)} \in \mathcal{T}, \forall x_{i,j}^{(k,f^k)} \in \{0, 1\}, i \leq n, j \leq m$, 文件压缩比 $\alpha = \{\alpha_{f_1}, \alpha_{f_2} \dots \alpha_{f_r}\}$, 文件大小集合 $\beta = \{\beta_{f_1}, \beta_{f_2} \dots \beta_{f_r}\}$, 新到达的请求 T_{fc}

- 1 $B_i = \sum_{b \in \mathcal{T}} \sum_{j=1}^m \alpha[f^b] * \beta[f^b] * x_{i,j}^{(b,f^b)}$, for $i \leq n$ 并且机器 b 存储 T_{fc} 的数据块;
 - 2 对 B 采用非降序排列;
 - 3 选择前 k_{fc} 个数据块, 并且把相应的 $x_{i,j}^{(c,f^c)}$ 设置为 1, 其他的设置成 0, 然后把所有的 $x_{i,j}^{(c,f^c)}$ 添加到 θ_{fc} ;
 - 4 **return** θ_{fc} ;
-

算法5中, 第 1 行计算入端口的负载。然后第 2 行对负载进行非降序排序, 第 3 行选择负载最小的 k_{fc} 源并且把 $x_{i,j}^{(c,f^c)}$ 设置成 1。对其他的端口的这个值设置为 0。最后, 把所有的 $x_{i,j}^{(c,f^c)}$ 加到 θ_{fc} 集合中, 然后返回 θ_{fc} 。

算法5尝试从所有的源节点中选择负载最小的源节点。认为这是有意义的, 因

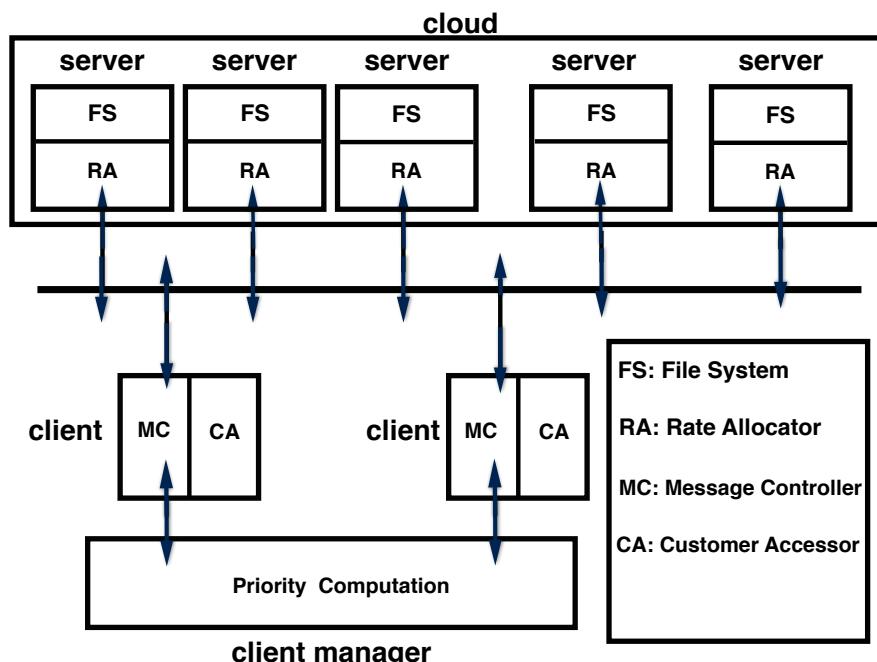


图 7.3 D-Target 系统设计

为，文件的获取时间 FAT 是由传输最慢的数据块决定的，因为 FAT 是由传输最慢的数据块决定的，因此，最小负载优先的策略可以减少文件平均获取时间。

7.4.4 D-Target 的设计

为了实现对就删码存储系统的调度，本文设计 D-Target。图7.3展示了 D-Target 的结构图。D-Target 有三个主要部件，客户端管理器，客户端信息收集器，服务器速率分配器。D-Target 采取集中式的调度器，调度器尝试最小化文件平均访问时间 (AFAT)。因为，服务器端的负载可以通过客户端的信息进行计算，所以调度器只需要对客户端进行管理。

Algorithm 6: 客户端管理器的操作过程

Input: 活跃的请求集合 \mathcal{T} , 为文件已经选好的集合 $\theta_{f^k}^{(k)} = \{x_{1,1}^{(k,f^k)}, x_{1,2}^{(k,f^k)} \dots x_{i,j}^{(k,f^k)} \dots\}, \forall T_{f^k}^{(k)} \in \mathcal{T}, \forall x_{i,j}^{(k,f^k)} \in \{0, 1\}, i \leq n, j \leq m$, 文件压缩比 $\alpha = \{\alpha_{f_1}, \alpha_{f_2} \dots \alpha_{f_r}\}$, 文件大小集合 $t\beta = \{\beta_{f_1}, \beta_{f_2}, \dots \beta_{f_r}\}$, 新到达的请求 T_{fc}

Output: 所有活动数据块的带宽

- 1 根据算法 4, 对 \mathcal{T} 和新来的请求 $T_{f^k}^{(k)}$ 进行排序;
 - 2 根据算法 7 给 \mathcal{T} 分配带宽;
 - 3 对所有端口的任务 \mathcal{T} 分配剩余的带宽;
 - 4 发送 $b_{i,j}$ 给对应的服务器;
-

Algorithm 7: 带宽计算的过程

Input: 已经排好序的集合 γ , 端口集合的剩余带宽集合 $Rem(\cdot)$

Output: 每个数据块的剩余带宽, 端口带宽剩余能力集合 $Rem(\cdot)$

- ```

1 for $k \in \gamma$ do
2 根据 (7-8) 计算负载 $g^{(k)}$;
3 for $t_{i,j}^{(k,f^k)} \in k$ do
4 $b_{i,j} = t_{i,j}^{(k,f^k)} / g^{(k)}$;
5 $Rem(P_i^{(in)}) - = b_{i,j}$;
6 $Rem(P_j^{(out)}) - = b_{i,j}$;
7 return 每个数据块的带宽, 端口剩余带宽集合 $Rem(\cdot)$

```
-

#### 7.4.4.1 客户端管理器

在 D-Target 中，客户端管理器是系统的核心，当有请求到达时，它计算请求的优先级 (Algorithm 4) 然后选择合适的源 (Algorithm 5)。计算完每个文件请求的优先级，D-Target 计算每个数据块的带宽，客户端管理器方法如算法6所示。在算法6中，当一个新的文件请求到达时，客户管理器对所有活动的请求根据算法 4(行 1) 进行排序 (行 2)。最后，如果链路还存在剩余带宽，那么把剩余带宽分给数据块，然后把相应的结果发给对应的服务器 (第 3 行 ~ 第 4 行)。

$$g^{(k)} = \max\left(\max_i \frac{\sum_{j=1}^m t_{i,j}^{(k,f^k)}}{Rem(P_i^{(in)})}, \max_j \frac{\sum_{i=1}^n t_{i,j}^{(k,f^k)}}{Rem(P_j^{(out)})}\right) \quad (7-8)$$

算法 7展示了带宽计算的细节。对每个已经排好序的请求，(7-8) 计算出瓶颈链路的完成时间 (第 2 行)。对于其他文件的数据块，使用瓶颈链路数据块的完成时间当做这个文件传输的数据块 (第 4 行)。最后，更新每个端口的剩余带宽 (第 6 行 ~ 第 7 行)。

#### 7.4.4.2 客户端

客户端存储文件的信息，包括数据块的位置，大小等。客户端和用户进行交互，并且收集文件请求的信息，最后，把这些信息发送给客户端管理器。服务器端存储所有的数据块，他们接受来自客户端获取文件的请求，然后根据客户端计算的结果来控制每条数据流发送的带宽。

### 7.5 实验验证

在本节中，通过真实使用真实的数据中心的流量来彻底评估 D-Target 的性能。本节的主要结果可以概述为：

(1) 使用 AT&T 的数据中心的流量，发现 D-Target 性能比 TCP, Aalo<sup>[39]</sup>, Barrat<sup>[36]</sup> 和 pFabric<sup>[32]</sup> 分别提高 2.5×, 1.7×, 1.8×, 3.6×。

(2) 不使用最小负载优先 (Smallest-Load-First, 简称 SLF) 启发式源选择，D-Target 性能分别比 TCP, Barrat, Aalo, pFabric 高 1.9×, 1.4×, 1.6×, 2×。以最小负载优先 (Smallest-Load-First, 简称 SLF) 启发式源选择，TCP, Barrat<sup>[36]</sup>, Aalo<sup>[39]</sup>, pFabric<sup>[32]</sup> 比没有 SLF 的方法效果好 30%, 27%, 33%, 20%。

(3) 离线场景下，与 2-近似的离线算法相比，在线算法的性能损失少于 15%。

### 7.5.1 仿真测试

本部分在大多数实验中使用 AT&T 中心的流量。这些数据流量是从 AT&T 的数据中心的 60 个机架的 720 个服务器收集来的，包括文件大小，数据块的大小，每个文件的块位置以及 30 天的请求。

**评价指标。** 使用平均文件访问时间 (Average File Access Time, 简称 AFAT) 来评估不同调度方法的有效性。考虑文件的特定类别的平均 FAT。FAT 涵盖了从调度程序的时间选择源到客户端接收所有的块。将 D-Target 与四个典型的比较调度算法：TCP，Aalo<sup>[39]</sup>，Barrat<sup>[36]</sup> 和 pFabric<sup>[32]</sup>，使用 TCP 作为基准，相对于 TCP 的性能提高定义为  $= \frac{\text{baseline of avg FAT}}{\text{current avg FAT}}$

**开源。** 为了使本部分实验可复原，发布了 D-Target 的主要代码。D-Target 的代码可以在<sup>[73]</sup> 下载。

### 7.5.2 真实流量仿真

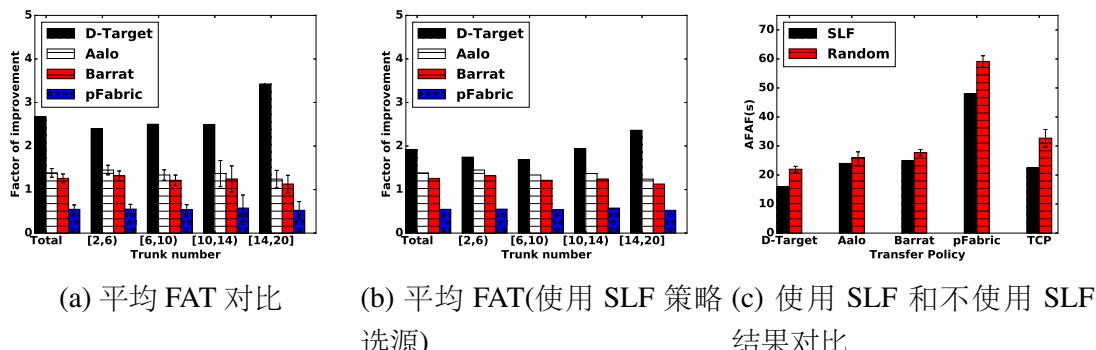


图 7.4 使用 AT&T 流量的实验结果，使用 TCP 当做基准。注意，对于一些随机选取的结果，每次实验的结果会不同，为了弥补这些不同，每组实验运行 100 次，取每次的平均值

在这一部分，用 AT&T 的纠删码系统的数据来测试 D-Target 的性能。对于源选择，因为每次源可能不相同，而 SLF 选取的源相同。为了消除随机的源选择出现的波动，每组实验运行 100 次，图7.4显示了实验结果。注意，在每组的实验结果中，error bar 表示的是平均值，最大值，最小值。

图7.4 (a) 显示了 AFAT 的结果。可以看到，D-Target 相对 TCP 提高  $2.5\times$ ，而对于 Aalo，Barrat，pFabric 相对 TCP 提高分别是  $1.5\times$ ， $1.2\times$ ， $0.8\times$ 。对于分布式纠删码存储系统，响应包含并行数据块，根据响应的宽度将其分为四组。当并行块数分别为 [2,6]，[6,10]，[10,14] 和 [14,20] 时，可以看到 D-Target 相对于 TCP 的改进倍数分别为  $2.2\times$ ， $2.4\times$ ， $2.8\times$ ， $3\times$ 。Aalo 相对于 TCP 的改进倍数分别为  $1.5\times$ ， $1.4\times$ ， $1.42\times$ ， $1.5\times$ 。Barrat 相对于 TCP 的改进倍数分别为  $1.3\times$ ， $1.5\times$ ， $1.2\times$ ， $1.15\times$ 。

pFabric 相对于 TCP 的改进倍数分别为  $0.6\times$ ,  $0.53\times$ ,  $0.6\times$ ,  $0.7\times$ 。随着块并行度的增加, 改善的幅度变得更大。这是因为, 并行块数越多, 请求之间的冲突越高, 所以进行源选择和流量控制的必要性越大。

在现代的纠删码存储系统中, 当请求到达时, 调度器总是使用随机源选择来选择那些源。在本文中, 以最小负载优先算法来选择源。在此情形下, 会发生较少的碰撞。图7.4 (b) 显示了所有传输方法使用最小负载优先的启发式来选择信号源的结果。可以看到 D-Target 和其他方法之间的差距变小了。平均而言 D-Target, Aalo, Barrat 比 TCP 分别提高  $1.9\times$ ,  $1.5\times$ ,  $1.3\times$ 。当并行块数分别为 [2,6], [6,10], [10,14] 和 [14,20] 时, 可以看到 D-Target 相对于 TCP 的改进倍数分别为  $1.8\times$ ,  $1.7\times$ ,  $1.9\times$ ,  $2.2\times$ 。Aalo 相对于 TCP 的改进倍数分别为  $1.2\times$ ,  $1.3\times$ ,  $1.4\times$ ,  $1.5\times$ 。Barrat 相对于 TCP 的改进倍数分别为  $1.1\times$ ,  $1.4\times$ ,  $1.3\times$ ,  $1.4\times$ 。

图7.4 (c) 显示了不同方法的平均文件访问时间 (FAT) 比较。可以看到, D-Target, Aalo, Barrat, pFabric, TCP 使用最小负载优先策略进行源选择分别比随机源选择性能提高大约 30%, 30%, 27%, 33%, 20%。

### 7.5.3 不同设置下性能

在这一部分, 探讨 D-Target 在不同参数下的性能。知道 FAT 是受源选择和数据块传输影响的。对于分布式纠删编码存储系统, 采用  $(n, k)$  MDS 编码进行编码, 其中  $n$  表示文件编码的块数,  $k$  表示重建文件所需的块数,  $k$  和  $n$  决定文件传输的宽度。因为要探索每个参数对系统性能的影响, 所以在每组实验中, 应该修正其他参数的值, 只改变想研究的因素。

表7.1显示了每个实验的默认设置。在默认参数中有 512 个文件, 文件大小在 100KB 到 10GB 之间。MDS 的默认参数值是 (8,4), 每个请求的到达时间在 0 到 1000s 之间。实验中的入口和出口端口容量是 1GB。对于每组实验, 生成参数 100 次, 绘制出最大值, 最小值, 平均值。

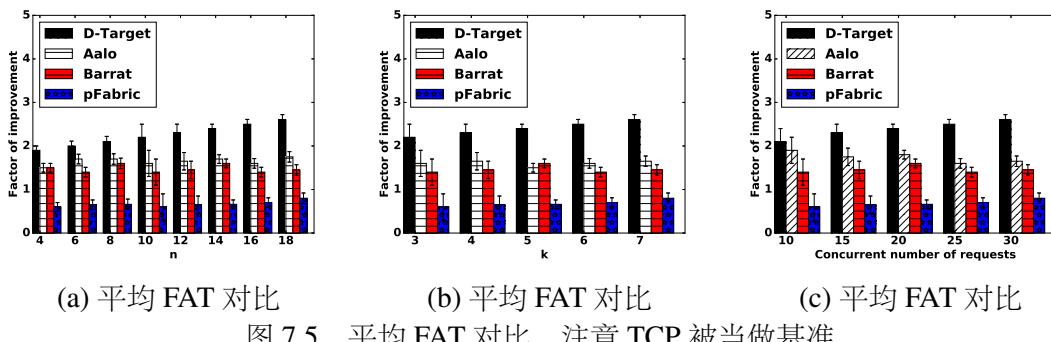


图 7.5 平均 FAT 对比, 注意 TCP 被当做基准

表 7.1 各组实验的缺省参数

| 请求的数量 | 文件的大小        | n | k | $\alpha$ | 链路容量 | 请求到达时间     |
|-------|--------------|---|---|----------|------|------------|
| 512   | [100KB,10GB] | 8 | 4 | 1.5      | 1GB  | [0s,1000s] |

图7.5 (a) 显示随着 n 的增加, D-Target 相对于 TCP 提高的幅度变大。当 n = 4 时, D-Target 相对于 TCP 提高 1.8×, Aalo 是 1.5×, Barrat 相对于 TCP 提高 1.5×, 当 n = 6 时, D-Target 相对于 TCP 提高的幅度是 1.9×, Aalo 相对于 TCP 提高的幅度是 1.6×, Barrat 相对于 TCP 提高的幅度是 1.4×, 当 n = 8 时, D-Target 相对于 TCP 提高的幅度是 2.0×, Aalo 相对于 TCP 提高的幅度是 1.7×, Barrat 相对于 TCP 提高的幅度是 1.5×。而当 n = 18 时, D-Target 相对于 TCP 提高的幅度是 2.5×, Aalo 是 1.5×, Barrat 相对于 TCP 提高的幅度是 1.3×。可以看到, D-Target 在更大的 n 值时表现更好, 这是 n 越大, 节点之间的冲突变得更高, 所以需要更好地优化源选择和更高效的传输。

图7.5 (b) 显示随着 k 的增加, D-Target 性能更好。当 k = 3 时, D-Target 相对于 TCP 提高的幅度 2.1×, Aalo 相对于 TCP 提高的幅度是 1.5×, Barrat 相对于 TCP 提高的幅度是 1.6×, 当 k = 4 时, D-Target 相对于 TCP 提高的幅度是 2.2×, Aalo 相对于 TCP 提高的幅度是 1.6×, Barrat 相对于 TCP 提高的幅度是 1.7×, 当 k = 7 时, D-Target 相对于 TCP 提高的幅度是 2.5×, Aalo 相对于 TCP 提高的幅度是 1.52×, Barrat 相对于 TCP 提高的幅度是 1.72×。k 越大时, D-Target 性能更好, 因为, k 越大, D-Target 由于较少的冲突而性能提高。

实际上, 实际系统中请求可以随时到达。请求到达时间可能会影响数据中心网络中的网络负载。图7.5 (c) 显示出了输并发对系统性能的影响。当并行传输的数据块数目为 10 时, D-Target 相对于 TCP 提高的幅度 2.1×, Aalo 相对于 TCP 提高的幅度是 1.9×, Barrat 相对于 TCP 提高的幅度是 1.5×, 当并行传输的数据块数目为 15 时, D-Target 相对于 TCP 提高的幅度是 2.2×, Aalo 相对于 TCP 提高的幅度是 1.8×, Barrat 相对于 TCP 提高的幅度是 1.6×, 而当并行传输的数据块数目为 30 时, D-Target 相对于 TCP 提高的幅度是 2.5×, Aalo 相对于 TCP 提高的幅度是 1.54×, Barrat 相对于 TCP 提高的幅度是 1.52×。可以看到, 对于较大的并发请求数量, D-Target 的性能提高不断增加。这是因为网络负载越重, 根据优先级调度的必要性增加。

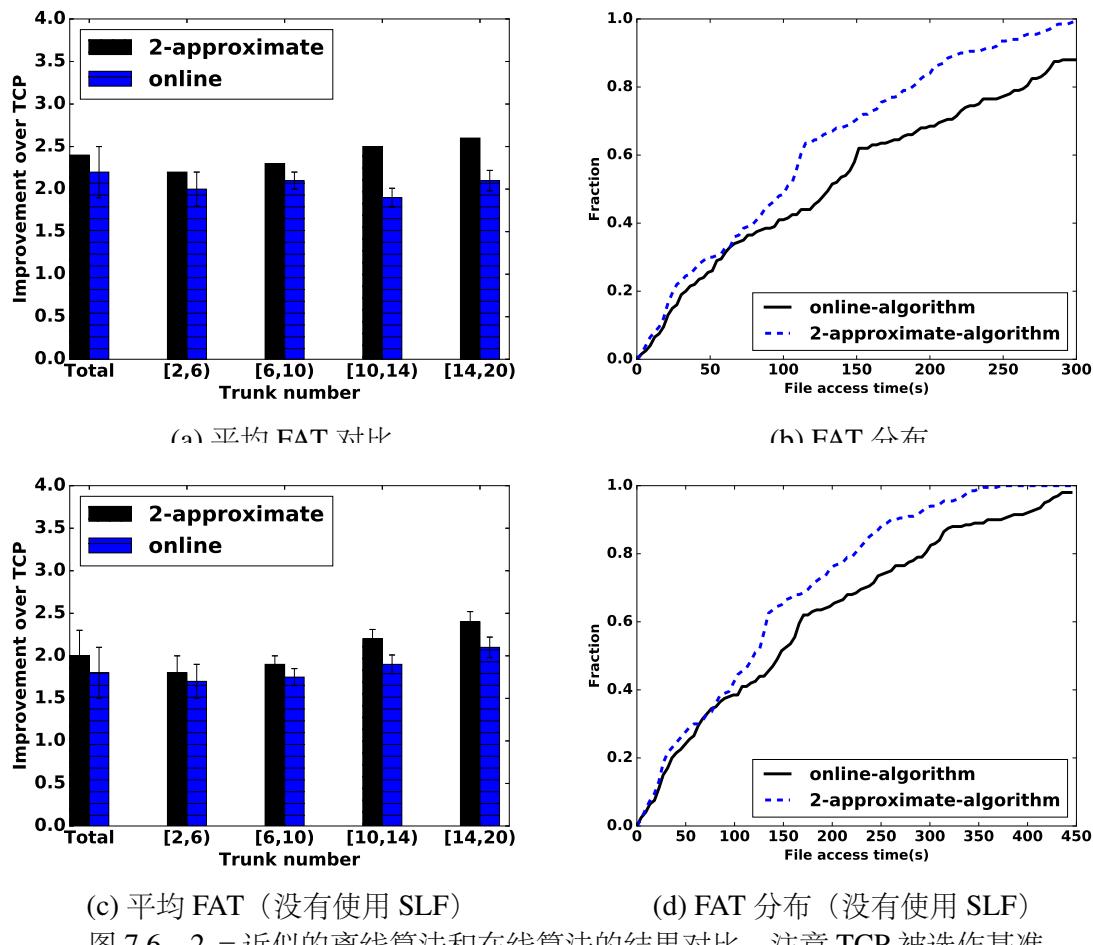


图 7.6.2 – 近似的离线算法和在线算法的结果对比。注意 TCP 被选作基准。

#### 7.5.4 离线算法和在线算法之间的差异

与离线的 2-近似算法相比，实时调度的在线算法 4 忽略负载的差异性，这可能导致性能损失。在本节中，研究在线算法和 2-近似离线算法之间的性能差距。使用 AT&T 的流量，并设置所有的请求到达时间为  $t = 0$ ，图 7.6 显示实验结果。

从图7.6（a）中可以看出，当并行块数分别为[2,6], [6,10], [10,14]和[14,20]时，2-近似方法的相对于TCP提高幅度是 $2.2\times$ ,  $2.3\times$ ,  $2.43\times$ ,  $2.52\times$ ，而在线方法提高幅度是 $2.1\times$ ,  $1.8\times$ ,  $1.9\times$ ,  $2.2\times$ 。平均起来2-近似方法的相对于TCP提高幅度是 $2.4\times$ ，而在线方法提高幅度是 $2.1\times$ 。在线方法有大约15%的性能损失。

图7.6 (b) 显示了FAT的分布情况,可以看出,两种近似方法中超过80%的FAT小于300s,而在线方法的FAT大于65%。在线方法有大约16%的性能损失。

图7.6 (c) 显示了没有最小负载优先的启发式的两种方法的性能对比，可以看出，当并行块数分别为 [2,6], [6,10], [10,14] 和 [14,20] 时，2-近似方法的相对于 TCP 提高幅度是 1.6×, 1.7×, 1.8×, 1.9×，而在线方法提高幅度是 1.6×, 1.6×, 1.7×, 1.8×。平均起来 2-近似方法的相对于 TCP 提高幅度是 2.0×，而在线方法提

高幅度是  $1.6\times$ 。在线方法有大约 20% 的性能损失。

图7.6 (d) 显示了 FAT 的分布，可以看到，不使用 SLF 情形下，在线的调度方法大约 80% 的 FAT 在 350s 以内，这比 SLF 的方法差 28% 左右。

## 7.6 本章总结

本章中提出了基于流信息的任务级传输优化方法对任务级的传输进行优化，并且提出最小化负载有限的启发式方法对纠删码存储系统进行访问优化。本章的优化目标是最小化文件平均访问时间（Average File Access Time，简称 AFAT）。基于此，本章设计并实现了 D-Target，一个集中式的调度器，并通过真实数据中心流量来评估 D-Target 的性能。

## 第8章 基于重要性和网络拥塞的任务传输调度方案

传统的网络资源管理机制主要是流级别或者包级别。最近，流组（coflow）作为一种新的并行应用数据传输通信模型而被提出。流组（coflow）对网络资源的应用级语义进行了有效的建模，因此可以通过将流组（coflow）作为网络资源分配或调度的基本元素来更好地实现一些高级优化目标，如减少应用的传输延迟等。虽然有效的流组（coflow）调度方法已经被研究，本章节中，建议对 coflow 调度时考虑权重，权重被用来表示不同 coflows 优先级或者 coflow 的紧急程度。本章引入加权流组完成时间（Weighted Coflow Completion Time，简称 WCCT）问题，设计一个不需要知道流组（coflow）信息就可以进行有效调度的算法，根据权重和网络拥塞程度调度流组（coflow），同时设计一个名为 Yosemite 的调度算法，并通过数据中心的真实流量对 Yosemite 进行仿真评测。评估结果显示，与最新的 coflow 调度算法相比，Yosemite 可以使得平均 WCCT 减少超过 40%，对于重要性在平均以上的 coflow，coflow 的平均完成时间减少超过 30% 的。和最有效的 coflow 调度方法相比，WCCT 优化性能大约提高了 30%，对于重要性在平均以上的 coflow，平均完成时间减少大约 25% ~ 30%。

### 8.1 概述

目前，数据中心很多实时的应用需要低延迟<sup>[4,60]</sup> 和高吞吐量<sup>[54]</sup>，例如那些使用 map-reduce 计算模型的密集型计算应用<sup>[72]</sup>，以及那些使用分布式文件存储<sup>[68,69]</sup>系统等。为了满足这些要求，数据中心网络基础设施已经专门的定制和改进，在拓扑设计，路由方案以及传输优化方面付出了巨大的努力。

在已有的各类成果中，流级别调度方法试图根据相应流的特征调度到达的流的数据包。例如，PDQ<sup>[28]</sup> 和 pFabric<sup>[32]</sup> 实现最短工作优先（Shortest Job First，简称 SJF）策略，以使短流抢占长流的带宽。因此，流平均完成时间（Average Flow Completion Time，简称 AFCT）减少，应用程序的传输延迟降低。但是，对于分布式应用包括很多并行的数据流，单条数据流的传输完成，并不能代表应用传输的完成，当并行数据流均传输完成时，才算传输完毕。因此只对单条流的传输优化，而不考虑它们之间的相互关系，应用的整体传输性能可能无法有效改善，甚至可能受到影响。

最近，流组（coflow）作为一种新的并行应用数据传输通信的抽象模型而提出。coflow 是两组机器之间流的集合，这些数据流具有相近的语义和类似的目标<sup>[38]</sup>。

例如，最小化集合中最慢的流的完成时间，或者确保集合中的流满足共同的期限。流组（coflow）对网络资源使用的应用级语义进行了有效的建模，因此可以通过将 coflow 作为网络资源分配或调度的基本元素来更好地实现高级优化目标，如减少应用程序的传输延迟。

最近很多成果已经研究了用于最小化流组（coflow）完成时间（Coflow Completion Time，简称 CCT）的高效 coflow 调度方法。Varys<sup>[37]</sup> 提出了最小有效瓶颈优先（Smallest- Effective-Bottleneck-First，简称 SEBF）启发式算法来确定 coflow 的调度顺序，并使用最小化期望分配带宽（Minimum-Allocation-for-Desired-Duration，简称 MADD）来计算要分配的带宽。然而，Varys 是一种智能程度很低的调度，它依赖于调度器得知流信息（如流大小或流到达时间）来决定如何调度流组（coflow）。这些限制会限制 Varys 在实际中的使用和部署。为了解决这个问题，新的一些研究比如 Aalo<sup>[39]</sup>，Barrat<sup>[36]</sup>，sunflows<sup>[59]</sup> 和 CODA<sup>[40]</sup> 可以不用预先得知 coflow 的信息（流大小或流到达时间）对 coflow 进行智能调度。

在本文中，对 coflow 的调度进行进一步的深入研究，建议对 coflow 赋予权重，分配给 coflow 的权重来表示 coflow 的优先级或者 coflow 应用的优先级。事实上，在实际中，应用有不同的重要性或者优先级，例如，分布式搜索应用的内部传输的数据流比支持分布式存储系统上的文件备份的数据流更为重要。当调度两个应用程序的 coflow 时，需考虑 coflow 的重要性。为此，给 coflow 分配不同的权重，即搜索引擎的 coflow 具有较高的权重，而文件备份的 coflow 具有较低的权重，的优化目标是最小化 coflow 的加权完成时间之和。

将非抢先式调度算法改造成不需要预先得知 coflow 信息的在线调度方案。在线算法 Yosemite 存在较小的性能损失，但可以在线调度 coflow，而不必事先知道 coflow 的大小或者到达时间。通过使用数据中心真实流量来仿真和测试评估来测试 Yosemite 的性能，并将其与具有最佳已知性能的最新的 coflow 调度算法（如 Varys<sup>[37]</sup>，Aalo<sup>[39]</sup> 和 Barrat<sup>[36]</sup> 进行对比。结果表明，Yosemite 在降低加权混合完成时间方面表现相当好。本章节做了以下工作：

- (1) 提出使用权重 coflow 作为数据中心资源调度的管理方案来给应用分配资源。从一个中型数据中心收集实际应用的流量，并深入分析权重 coflow 调度的必要性和重要性。
- (2) 评估 coflow 的各个调度策略并与最新的不用预先得知 coflow 的信息调度的算法对比。使用真实的数据中心流量评估并且测试 Yosemite，评估结果发现，Yosemite 可以减少超过 40% 的 WCCT。此外，对于超过平均水平的 coflow 而言，Yosemite 可以减少平均完成时间大约 30%。和当前最有效的不可预知 coflow 进行

调度的方法相比，Yosemite 分别减少了约 30% 的平均 WCCT 和对紧急程度在平均值以上的 coflow 的 20%~30% 的完成时间。

## 8.2 研究动机和相关工作

数据中心现在正成为托管大量服务和应用程序的重要基础设施。为了满足数据中心中应用对高吞吐量带宽和低延迟的需求，大量的研究工作致力于网络资源分配和调度。例如，DCTCP<sup>[16]</sup>，D<sup>2</sup>TCP<sup>[23]</sup>，L<sup>2</sup>DCT<sup>[24]</sup>，LPD<sup>[57]</sup>，D<sup>3</sup><sup>[27]</sup> 和 PDQ 是流级别的速率控制或调度方案，这些方案侧重于优化流的完成时间，或者满足流对期限的要求。

Coflow 的概念最近已经被提出<sup>[38]</sup>，以满足更高层次应用对性能要求。其中 coflow 是具有相同的语义和共同目标的两组机器之间的流集合。当前业界关心最小化 coflow 平均完成时间。特别是，Varys 是基于网络瓶颈进行的优化，Varys 使用最小瓶颈优先的启发式算法来决定 coflow 的优先级，最终优化 coflow 的平均完成时间。而 D-CLAS<sup>[41]</sup>，Aalo<sup>[39]</sup>，sunflows<sup>[59]</sup> 和 CODA<sup>[40]</sup>，试图“猜测”coflows 的一些特性，智能化的对 coflow 进行调度。前者被称为“白盒”方法，因为这种方法需要预先得知 coflow 的大小，宽度的一些信息，后者被称为“黑盒”方法，因为调度时，不需要提前得知 coflow 的信息，系统会自动的获取和识别这些信息。

虽然这些 coflow 级别的调度方法确实提高了通信传输的性能和效率，但是这些方法把 coflow 都看作同样重要。coflow 或者应用之间调度的性能差异取决于不同的调度策略。例如，有的调度方案倾向于短的 coflow 优先，有的调度方案倾向于宽的 coflow 优先，不同的方案达到不同的调度效果。从此角度，认为 coflow 或者应用是有优先级，当调度 coflow 或者应用时，优先级应该被考虑进去。例如，分布式搜索应用的内部传输的数据流，比支持分布式存储系统上的文件备份的数据流更为重要。当前对 coflow 的调度策略，主要侧重根据 coflow 的宽度和长度对 coflow 进行优化，应用的重要性和 coflow 的长度以及宽度是没有相关关系。因此，当前出现的很多调度算法，并不能满足 coflow 调度的核心需求。

为了说明数据中心内 coflow 重要性应该被考虑，对来自一个中型的数据中心的流量进行测量和分析，假设数据中心中有 3000 台机器，同时并行了 100 个应用程序。在和数据中心管理人员以及网络工程师进行深入讨论以后，深入分析了从 60 个机架中收集的 720 个服务器的流量信息。流持续 1 个月左右，表 8.1 显示的是最常用的前 10 个应用的信息。把应用分成 5 个重要级：紧急的应用，重要的应用，正常的应用，不重要的应用和松散的应用。例如，事件程序产生的流量和 vRoute 应用程序产生的路由信息流量，具有最高的优先级。而数据备份和数据分发是在

表 8.1 数据中心的应用和它们的紧急程度

| App Name    | Type          | width | length (MB) | Emergence Level |
|-------------|---------------|-------|-------------|-----------------|
| Event       | communication | 20    | 5           | significant     |
| vRouter     | communication | 8     | 3           | significant     |
| Druid       | interactive   | 6     | 18          | important       |
| Hadoop      | computation   | 5     | 42          | normal          |
| Web         | interactive   | 3     | 5           | normal          |
| VoltDB      | background    | 4     | 21          | normal          |
| Hive        | background    | 7     | 32          | unimportant     |
| Redies      | background    | 2     | 30          | unimportant     |
| data-backup | background    | 3     | 124         | lax             |
| data-dist   | background    | 5     | 93          | lax             |

表 8.2 Hadoop 内部的 coflow 和它们的重要性

| Coflow Type  | width | length (MB) | Emergence Level |
|--------------|-------|-------------|-----------------|
| index-sort   | 10    | 3           | significant     |
| db-analysis  | 3     | 12          | important       |
| index-count  | 6     | 20          | normal          |
| log-analysis | 6     | 31          | unimportant     |
| crawler      | 4     | 12          | unimportant     |
| word-count   | 3     | 11          | unimportant     |

后台运行的应用程序，产生的是背景流量，背景流量的优先级较低。看到，应用的数据流小或者包含更宽的流，并不代表具有更高的优先级。例如，Hive（平均宽度是 7）比 Web（平均宽度是 3）的宽度大，但 Hive 比 Web 的优先级低。另一个例子是 Web（平均长度是 5MB）的平均长度比 Druid（平均长度是 18MB）小，但是 Web 的优先级比 Druid 低。这支持了前面的观点，即一般情况下，应用的 coflow 的紧急程度和宽度以及长度没有必然的关联。表 8.2 进一步显示了 hadoop 应用中不同 coflow 的细节，从中可以得出类似的结论。例如，word-count（平均宽度是 3）比 index-count（平均宽度是 6）的宽度小，优先级低。另一个例子是 log-analysis（平均宽度是 6）的平均长度比 db-analysis（平均宽度是 3）大，但是优先级低。这支持了前面的观点，即一般情况下，应用的 coflow 的紧急程度和宽度以及长度没有必然的关联。

为了说明现有的 coflow 调度方法的无效性，绘制了不同 coflows 的平均 Cflow 完成时间 (CCT)，按照紧急程度进行，如图 8.1 所示。图 8.1 (a) 显示了所有应用在使用 TCP 的情形下平均 Cflow 完成时间 (CCT)。可以看到，对于紧急的应用

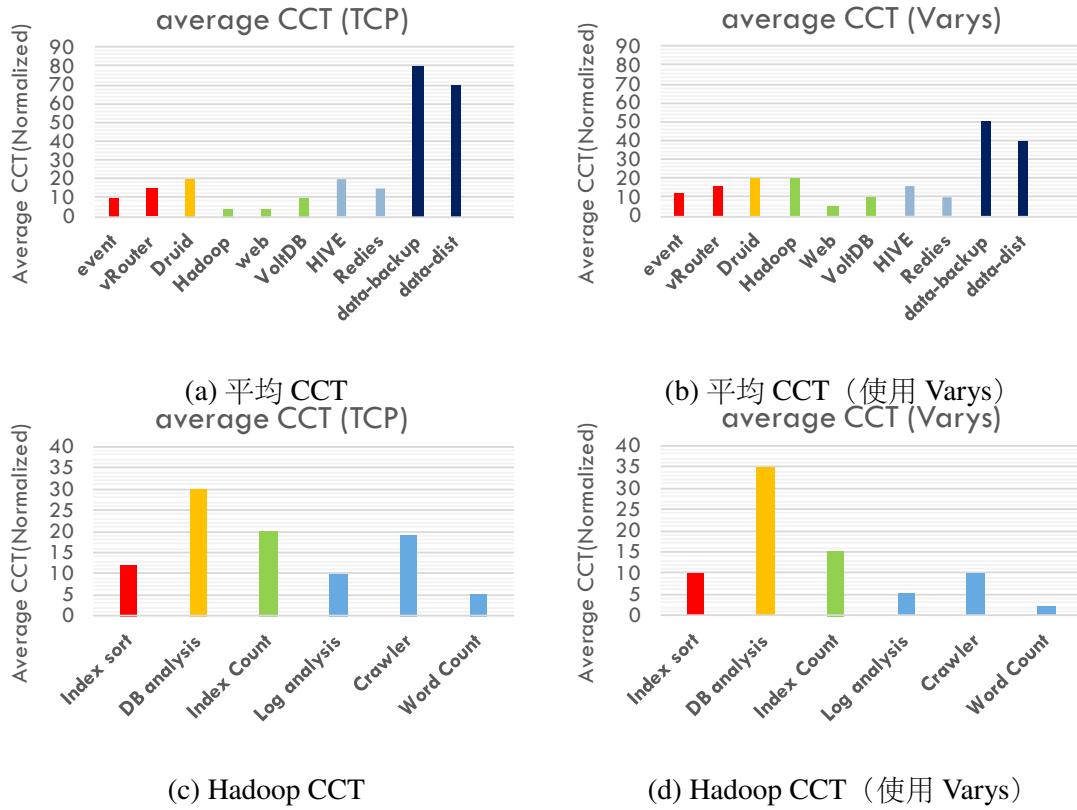
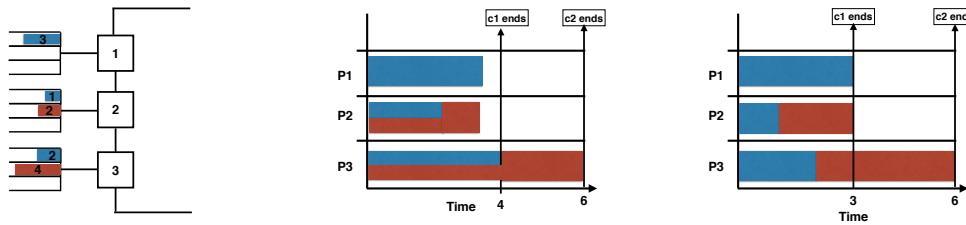


图 8.1 平均 Coflow 完成时间, coflow 根据紧急程度进行分类

图 8.2 两条 coflow 同时到达端口, coflow  $c_1$ (蓝色) 包含 3 条流, coflow  $c_2$ (红色) 包含 2 条流。 $c_2$  的优先级比  $c_1$  高

Event 和 vRouter, 平均 CCT 是 9, 11。Druid 的紧急程度是重要, 平均 CCT 是 20。Hadoop, web, VoltDb 的紧急程度是正常程度, 平均流组完成时间是 6, 7, 10。Hive, Redis 的紧急程度是不重要, 平均流组完成时间是 20, 15。data-dist 和 data-backup 的紧急程度是松散, 平均流组完成时间是 80, 70。而图8.1 (b) 显示了使用 Varys 进行调度时的仿真结果。对于紧急的应用 Event 和 vRouter, 平均 CCT 是 11, 15。Druid 的紧急程度是重要, 平均 CCT 是 21。Hadoop, web, VoltDb 的紧急程度是正常程度, 平均流组完成时间是 20, 5, 9。Hive, Redis 的紧急程度是不重要, 平均流组完成时间是 12, 10。data-dist 和 data-backup 的紧急程度是松散, 平均流组完成时间是 50, 40。通过比较图8.1 (a) 和图8.1 (b), 可以看到, 对于紧急程度

是紧急和重要的应用，**Varys** 对平均完成时间基本没有提高，甚至对于 Event，使用 **Varys**，平均流组完成时间甚至增大。**Varys** 对于紧急程度为正常，不重要和松散的应用提高幅度较大。

图8.1 (c) 显示的是对于 Hadoop 应用使用 TCP 的情形下平均流组完成时间 (Coflow Completion Time, 简称 CCT)。可以看到，对于紧急的应用 Index Count，平均 CCT 是 12。DB Analysis 的紧急程度是重要，平均 CCT 是 35。Log analysis, crawler, word-count 的紧急程度是不重要，平均 CCT 是 10, 20, 5。而图8.1 (d) 显示了使用 **Varys** 对 Hadoop 应用进行调度时的仿真结果。对于紧急的应用 Index Count，平均 CCT 是 10。Druid 的紧急程度是重要，平均 CCT 是 21。Hadoop, web, VoltDb 的紧急程度是正常程度，平均 CCT 是 5, 10, 3。通过比较图8.1 (c) 和图8.1 (d)，可以观察到和图8.1 (a), 图8.1 (b) 类似的效果。

为了分析上述问题出现的原因，展示了一个小的调度实例，如图8.2所示。在这个实例中，如图8.2(a) 所示，两条 coflow， $cf_1$  和  $cf_2$  同时到达端口， $cf_1$  有 3 条子流， $cf_2$  有 2 条子流， $cf_2$  的优先级比  $cf_1$  高。图8.2(b) 展示的是使用 TCP 作为调度协议的结果，图8.2(c) 展示的是使用 **Varys** 作为调度方法的结果。可以发现，相对于 TCP，更加紧急的  $cf_2$  完成时间未变，而  $cf_1$  减小了 1。分析出现这个问题的原因是 **Varys** 使用 **Smallest-Effective-Bottleneck-First** (SEBF) 但是忽略了 coflow 的重要性。这提示，在调度应用的数据任务时，coflow 的重要性应该也被考虑，使用权重表示 coflow 的重要性，优化目标位平均权重完成时间。

### 8.3 Yosemite 算法设计

本节提出基于重要性和网络拥塞的任务传输调度方案 – **Yosemite**，第6章提出的算法2中假设所有的 coflow 是同时到达，并且调度器没有预先得知 coflow 的信息，这些理想化的假设是不现实的，因此在设计能够真实部署的算法时，算法应该慎重的使用。

算法2的关键思想在于第 7 行，它试图在大多数加载的端口  $p^*$  上优先考虑具有权重大但负载小的 coflow。实际上，当前数据中心网络中的端口通过利用某些负载平衡技术来进行负载均衡<sup>[72]</sup>。所以做一个忽略端口负载差异的简化，并且把权重大但负载小的 coflow 设置为高优先级。通过用这种启发式算法替换算法2中的 coflow 优先级设定规则，随后得到一个简单的离线调度算法8。

**Algorithm 8:** 简单离线调度算法

---

**Input:** Coflow 集合  $\mathcal{F} = \{F^{(k)}\}$ , 权重集合  $\mathcal{W} = \{w_k\}, 1 \leq k \leq n$ ;

**Output:** a permutation  $\gamma$  of  $\{1, \dots, n\}$ ;

```

1 $\mathcal{R} \leftarrow \{1, \dots, n\}$;
2 for $r \in \mathcal{R}$ do
3 $L^r = \max(\max_i \sum_{j=1}^m f_{i,j}^{(k)}, \max_j \sum_{i=1}^m f_{i,j}^{(k)})$;
4 for i from 1 to n do
5 $\gamma[i] = r^* = \arg \max_{r \in \mathcal{R}} w_r / L^r$;
6 $\mathcal{R} = \mathcal{R} \setminus \{r^*\}$;
7 return γ ;

```

---

### 8.3.1 解决 IWCCM 的另一个简单离线算法

算法8展示的是一个简单的离线调度算法。算法8首先定义 coflow 的负载为传输最慢的 coflow 传输完成需要的时间。算法的第 2 行, 首先计算出所有 coflow 的负载, 因为在假设的 non-blocking 结构中, 所有端口的转发能力都是 1, 因此, 在非强占的调度场景下, coflow 的负载是最长的流的长度。第 4~6 行是对 coflow 进行排序, 简单的离线算法把权重大但负载小的 coflow 设置为高优先级。事实上, 算法8依然假设所有的 coflow 同时到达, 并且需要预先得知 coflow 的信息, 然而, 在真实的环境中, coflow 的到达是随机的, 并且, 对于一些应用如 hadoop 等, 很难预先得知要传输的数据流的信息, 因此, 实际中可以使用已发送的流的大小进行流的负载的计算, 发送的数据越多, 说明数据流越长, 可能的负载就越大在做出所有这些改变之后, 得到在线调度算法9。

### 8.3.2 Yosemite 算法

算法9是在线的调度算法, 当一个 coflow 到来时, 算法9被调度器调用一次。算法9根据上面提出的优先级排序策略对 coflow 进行排序 (行 1 ~ 11)<sup>①</sup>, 随后, 给每条流分配带宽 (行 12 ~ 19)。最后如果有剩余带宽, 分配这些剩余带宽给流, 这样保证网络资源最充分的利用。

算法9不需要预先知道 coflow 的信息, 它动态的根据权重和网络的情形给 coflow 分配带宽。事实上, 算法9近似实现最短 coflow 优策略。和算法8相比, 这

---

<sup>①</sup> 一个 coflow 的集合  $\mathcal{F}$  和对应调度顺序被调度器自始至终维护着, 如果没有新的 coflow 到达, 优先级排序策略会被忽略, 所有的 coflow 的顺序保持不变。

个算法不可避免的会存在性能损失，在后面的实验中，会介绍因为放缩而引起的性能损失。

---

**Algorithm 9:** 在线算法

**Input:** Coflow list  $\mathcal{F} = \{F^{(k)}\}$ , weight list  $\mathcal{W} = \{w_k\}$ ,  $1 \leq k \leq n$ ;

**Output:** a permutation  $\gamma$  of  $\{1, \dots, n\}$ ;

```

1 if a new coflow arrives then
2 update the coflow list $\mathcal{F} = \{F^{(k)}\}$;
3 $n = |\mathcal{F}|$;
4 for $1 \leq k \leq n$ do
5 update $s_{i,j}^{(k)}$, the cumulative volume of the traffic sent by coflow $F^{(k)}$, for
 $1 \leq i, j \leq m$;
6 $L_i^{(k)} = \sum_{j=1}^m s_{i,j}^{(k)}$ for $1 \leq i \leq m$;
7 $L_{j+m}^{(k)} = \sum_{i=1}^m s_{i,j}^{(k)}$ for $1 \leq j \leq m$;
8 $l^{(k)} = \max_{1 \leq i \leq 2m} L_i^{(k)}$;
9 $\alpha^{(k)} = l^{(k)} / w_k$;
10 ω = sort the list of $\{\alpha^{(k)}\}$ in nondecreasing order;
11 γ = index set of ω ;
12 initialize the bandwidth to be allocated on each ingress and egress port:
13 $I_p = O_p$ = port physical bandwidth, for $1 \leq p \leq m$;
14 for ℓ from 1 to n do
15 schedule coflow $F = F^{(\gamma[\ell])}$;
16 $r = \min \{I_p, O_q\}$, subject to F still need to send traffic to port p or receive
 traffic from port q ;
17 for each unfinished flow $F_{i,j}$ in F do
18 allocate bandwidth of r to $F_{i,j}$;
 update I_i and O_j by deducting r from them ;
19 allocate remaining bandwidth equally to all flows;

```

---

## 8.4 实验验证

在这个部分中，评估了在线和离线算法，首先，使用 facebook<sup>[37]</sup> 的真实流量对离线算法和在线算法进行评估；随后使用 AT&T 数据中心流量对算法进行评估；最后，本文对离线算法和在线算法进行评估，得到在线算法性能损失。本部分实验结论如下

- (1) 使用 facebook<sup>[37]</sup> 的真实流量（coflow 的权重是随机生成），Yosemite 在平均 coflow 权重完成时间上，性能比 Varys<sup>[37]</sup>, Aalo<sup>[39]</sup> 和 Barrat 提高 30%, 40%, and 50%，对于紧急程度在平均以上的 coflow，Yosemite 的性能比 Varys, Aalo, Barrat 分别提高 20% ,30%, 40%。
- (2) 对于测量的数据中心流量，和 varys（当前性能最好的 coflow 调度策略）相比，Yosemite 可以减小大约 30% 的 WCCT，并且对于紧急的 coflow，Yosemite 可以减小大约 25%-30% 的完成时间。
- (3) 和 2-近似的离线算法相比，的在线算法有不到 30% 的性能损失。

### 8.4.1 仿真方法介绍

本部分的仿真是基于两种数据中心真实流量，第 1 组流量是从 facebook 收集的 150 个机架上的 3000 台机器<sup>[37]</sup>。第 2 组流量是从一个中等规模的数据中心，在这个数据中心中，100 个应用同时在 60 个机架上的 720 台机器上运行。在实验中，流量根据紧急程度被分成了 5 个类别：紧急，重要，正常，不重要，松散，这 5 个类别的权重设置为 5, 4, 3, 2, 1。因为 facebook 的流量中，并没有权重的信息，因此，对于 facebook 的流量，随机给 coflow 分配权重。

对于策略性能的评估，使用两个指标，第一个指标是 coflow 完成时间 (Coflow Completion Time 简称 CCT)，第二个是权重 coflow 完成时间 (Weighted Coflow Completion Time，简称 WCCT)。考虑平均 CCT 或者所有所有 coflow 的权重完成时间，同时也考虑一些特殊 coflow 的完成时间和权重完成时间。用三个典型的 coflow 策略和 Yosemite 进行对比：Varys<sup>[37]</sup>, Aalo<sup>[39]</sup> and Barrat<sup>[36]</sup>。Varys 是需要知道 coflow 的各种信息的，是其中性能最优的。Aalo 不需要知道 coflow 的长度，Barrat 使用分布式的方式运行的。此外，有一些其它的 coflow 的调度方法，比如 sunflows<sup>[59]</sup> 和 CODA<sup>[40]</sup>。然而，sunflows 侧重于特殊的通信模型，CODA 使用机器学习的方法，性能不如 Aalo。为了进行对比，把 TCP 当作比较基准，比较相对于 TCP 提高的倍数，比如， $\frac{CCT \text{ by TCP}}{CCT \text{ by ascheduler}}$ ，和  $\frac{WCCT \text{ by TCP}}{WCCT \text{ by ascheduler}}$ 。为了进行深层次的对比，coflows 进一步的根据 coflow 的长度（最长 coflow 中流的长度）和宽度（coflow 中数据流的数目）分成 Narrow&Short (N-S), Narrow&Long (N-L), Wide&Short (W-S)，和

Wide&Long (W-L) 这 4 类进行区分。

#### 8.4.2 Facebook 流量下仿真测试

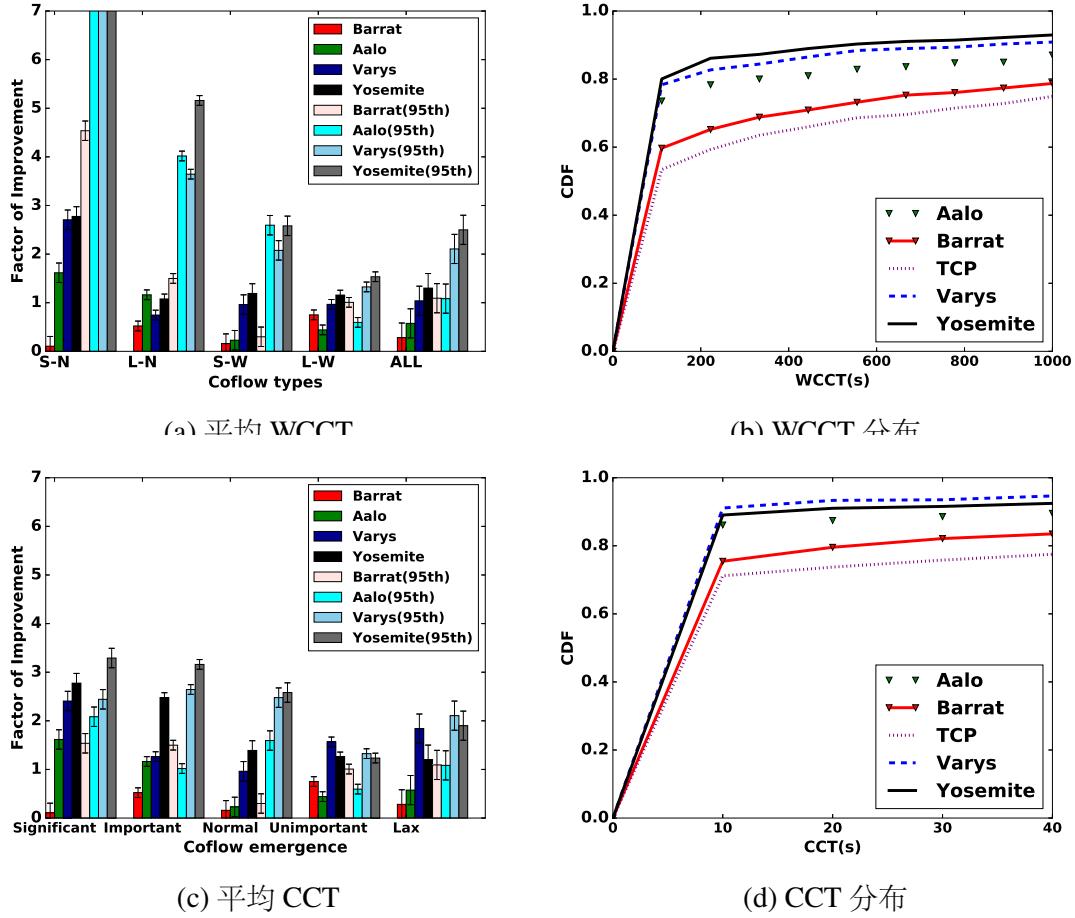


图 8.3 [仿真] 使用 facebook 数据, 平均 WCCT 和平均 CCT, TCP 被当作基准

本部分使用 facebook<sup>[37]</sup> 的流量来测试 Yosemite 的性能。Facebook 的原始的数据并不包含权重信息, 因此实验中随机给 coflow 在 1, 2, 3, 4, 5 中选择权重进行赋值。因为权重是随机产生的, 因此, 每组实验重复 100 次, 来避免随机产生权重带来的偶然影响。实验图中绘制了 error bar, 同时标记了最大值, 最小值以及平均值。为了消除一些极值的影响, 考虑更一般化的情形, 在每组中, 同时删除最前和最后 2.5% 结果, 在实验中同时绘制了 95% 的情形。

图8.3 (a) 显示的是 Yosemite 对 WCCT 优化的结果, 发现 Yosemite 对平均 WCCT 优化的性能是最好的。Yosemite 对于 TCP 性能提高的倍数为  $3.5 \times$  (N-S),  $1.6 \times$  (N-L),  $1.7 \times$  (W-S),  $1.4 \times$  (W-L) 和  $1.5 \times$  (ALL)。使用 Varys, 各个类型的 coflow 对于 TCP 性能提高的倍数为  $2.4 \times$  (N-S),  $0.9 \times$  (N-L),  $1.2 \times$  (W-S),  $1.3 \times$

(W-L) 和  $1.2 \times$  (ALL)。而 Aalo 对于 TCP 性能提高的倍数为  $1.6 \times$  (N-S),  $1.2 \times$  (N-L),  $0.3 \times$  (W-S),  $0.6 \times$  (W-L) 和  $0.8 \times$  (ALL)。使用 Barrat, 各个类型的 coflow 对于 TCP 性能提高的倍数为  $0.4 \times$  (N-S),  $0.8 \times$  (N-L),  $0.2 \times$  (W-S),  $0.8 \times$  (W-L) 和  $0.4 \times$  (ALL)。

为了避免一些极端情况对平均性能的影响, 图8.3 (a) 同时显示了 95% 的实验结果。使用 Yosemite, 各个类型的 coflow 对于 TCP 性能提高的倍数为  $25 \times$  (N-S),  $5.5 \times$  (N-L),  $2.5 \times$  (W-S),  $1.5 \times$  (W-L) 和  $2.5 \times$  (ALL)。使用 Varys, 各个类型的 coflow 对于 TCP 性能提高的倍数为  $22 \times$  (N-S),  $4.5 \times$  (N-L),  $2.1 \times$  (W-S),  $1.2 \times$  (W-L) 和  $2.1 \times$  (ALL)。使用 Aalo, 各个类型的 coflow 对于 TCP 性能提高的倍数为  $15 \times$  (N-S),  $4.1 \times$  (N-L),  $2.5 \times$  (W-S),  $0.5 \times$  (W-L) 和  $1.2 \times$  (ALL)。使用 Barrat, 各个类型的 coflow 对于 TCP 性能提高的倍数为  $4.5 \times$  (N-S),  $1.6 \times$  (N-L),  $0.2 \times$  (W-S),  $1.2 \times$  (W-L) 和  $1.1 \times$  (ALL)。

图8.3 (b) 表示 WCCT 的分布。可以看到, 80% 以上的 Yosemite 的 WCCT 在 200s 以内, 而 Varys, Aalo, Barrat 的是 70%, 65%, 60%。Yosemite 比 Varys, Aalo 和 Barrat 要好 15%, 25%, 30%。

图8.3 (c) 显示了具有不同紧急程度的 coflows 平均 CCT。可以看到 Yosemite 对于 TCP 性能提高的倍数是  $2.5 \times$  (紧急),  $2.3 \times$  (重要),  $1.2 \times$  (正常),  $1.4 \times$  (不重要),  $1.5 \times$  (松散)。Varys 是  $2.2 \times$  (紧急),  $1.5 \times$  (重要),  $1.4 \times$  (正常),  $1.6 \times$  (不重要),  $1.9 \times$  (松散)。Aalo 是  $1.2 \times$  (紧急),  $1.4 \times$  (重要),  $0.3 \times$  (正常),  $0.6 \times$  (不重要),  $0.7 \times$  (松散)。Barrat 是  $1.4 \times$  (紧急),  $1.2 \times$  (重要),  $0.3 \times$  (正常),  $0.8 \times$  (不重要),  $0.8 \times$  (松散)。图8.3 (c) 同时显示了 95% 的情形, 对于 95% 的场景, Yosemite 对于 TCP 性能提高的倍数是  $3.2 \times$  (重要),  $3.1 \times$  (重要),  $2.9 \times$  (正常),  $1.6 \times$  (不重要),  $1.9 \times$  (松散)。Varys 是  $2.2 \times$  (紧急),  $2.1 \times$  (重要),  $2.0 \times$  (正常),  $1.9 \times$  (不重要),  $2.3 \times$  (松散)。Aalo 是  $1.5 \times$  (紧急),  $1.2 \times$  (重要),  $0.6 \times$  (正常),  $0.8 \times$  (不重要),  $0.7 \times$  (松散)。Barrat 是  $0.2 \times$  (紧急),  $0.6 \times$  (重要),  $0.4 \times$  (正常),  $0.8 \times$  (不重要),  $0.5 \times$  (松散)。可以看到, Yosemite 对于紧急的和重要的 coflow 的优化水平要比 Varys 好 20%, 但对于重要性在正常以下的 coflow(包含正常), Varys 比 Yosemite 好 20%。这是因为当 Varys 调度 coflows 只考虑网络的情形, 并没有把 coflow 的重要性考虑进去。而 Yosemite 既考虑到网络条件, 又考虑到 coflows 的重要性。在 Yosemite 之下, 紧急的和重要的 coflow 有更高的优先级, 因此平均完成时间会缩短。

图8.3 (d) 显示了不同方法下的 CCT 的分布。Varys 下 80% 以上 CCT 在 20s 以内, Yosemite, Aalo, Barrat 分别为 75%, 65%, 60%。统计所有 coflow 的平均

CCT, Varys 比 Yosemite 的效果要好 10%。可以看到，Varys 比 Aalo, Barrat 在平均 CCT 和平均 WCCT 最小化方面表现更好。因此在下面的实验中，主要使用 Varys 作为参考方法来展示 Yosemite 的性能。

#### 8.4.3 使用中等规模的数据中心流量数据进行测试

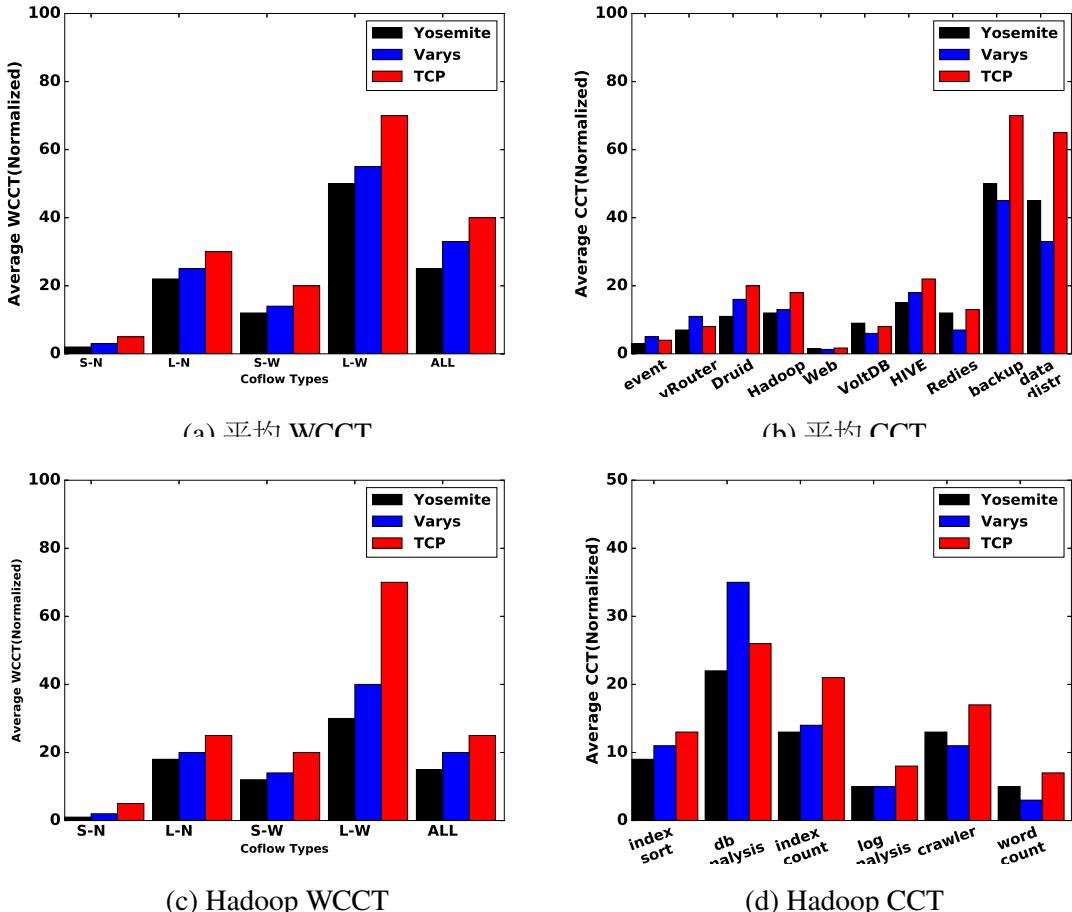


图 8.4 [仿真] 所有应用和 hadoop 的平均 CCT (标准化) 和平均。Varys 忽略 coflow 的重要性，只考虑网络拥塞，Yosemite 同时考虑 coflow 的重要性和网络拥塞

在这个部分中，解决8.2部分提出的问题。在的中等规模的数据总心中，应用有 5 个优先级，紧急，重要，标准，不重要，松散。使用 5, 4, 3, 2, 1 来表示这 5 个优先级。图8.4显示了实验结果。

图8.4 (a) 显示了对所有 coflows 的平均 WCCT。发现 Varys 的平均 WCCT 是 8 (Narrow&Short), 20(Narrow&Long), 15 (Wide&Short), 40 (Wide&Long) and 30(ALL)。Yosemite 的平均 WCCT 是 5 (Narrow&Short), 15(Narrow&Long), 10 (Wide&Short), 30 (Wide&Long) and 20(ALL)。对于最小化平均 WCCT，Yosemite 性能大约有 20% 的性能提升。

从图8.4(b)，看到对于 vRouter 和 event，这两个应用有紧急程度的应用，Varys 的性能表现和 TCP 基本相同，Yosemite 比 TCP 性能提高 20%。对于 Hadoop 和 Druid，这两个应用的优先级是重要，Yosemite 的性能比 Varys 高 30%。对于其它的应用，Varys 性能比 Yosemite 好 10%。

图8.4(c) 展示的是 Hadoop 平均 WCCT，图8.4(d) 展示的是 Hadoop CCT。对于 index sort 和 db analysis，这两种应用在数据中心中优先级属于重要的优先级，Varys 性能甚至比 TCP 差。然而，使用 Yosemite，性能大约有 30%~40% 提升。但是对于不重要和松散的 coflows，Varys 甚至性能更好。对于平均 WCCT，Yosemite 性能比 Varys 有 20% 性能提升。

#### 8.4.4 权重的讨论

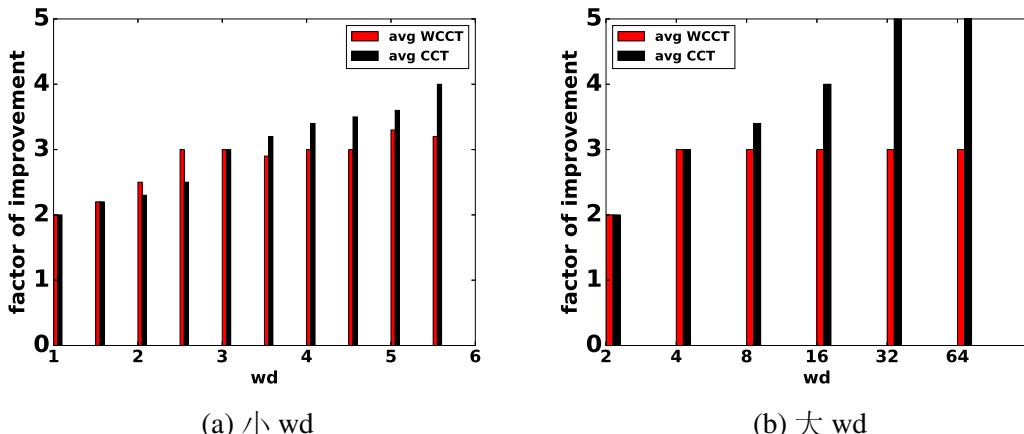


图 8.5 [仿真] 不同权重设置下的，平均 WCCT 和平均 CCT，TCP 被当做基准

权重在 Yosemite 调度系统中有重要的作用，在上面的实验中，权重之间的差是 1。在这组实验中使用 Facebook 的流量，设置权重在  $1, 1+wd, 1+2*wd, 1+3*wd, 1+4*wd$  之间，改变 wd 从 1 到 10，图8.5(a) 显示了平均 WCCT 相对于 TCP 提高的比例，重要 coflow ( $weight > 1 + 2 * wd$ ) 的平均 CCT 相对于 TCP 提高的比例。可以看到随着权重差异的增加，平均 WCCT 的提高幅度从 2 增加到 2.8，但是对于重要 coflow 的平均 CCT 提高幅度基本保持不变。原因是 Yosemite 使用权重和已经发送数据流的大小作为排序因子，随着权重差异的增加，权重扮演的角色越来越大，因此重要的 coflow 排序在前，所以优先获得更多的网络资源，进而先传输完成。图8.5(b) 展示的是更大的 wd 差异值，wd 从 2 增大到 64，看到当  $wd > 8$  时，重要性高得 coflow 提高变小，这和图8.5(a) 的结果基本类似。

在实际中，权重的设置可以根据网络管理员的需求来进行决定，当管理员认为 coflow 的紧急程度是十分重要的因子时，可以用比较大的 wd，当管理员认为网络拥塞情形和 coflow 的重要性同等重要时，可以使用小的 wd。

#### 8.4.5 策略和最优策略的差距

找到 IWCCM 问题的最优解很困难，但是找到了一个 LP-based 的策略<sup>[45]</sup>，这个算法的近似度为  $\frac{67}{3}$ ，用 LP-based 策略和 Yosemite 策略进行对比。使用 facebook 的流量，并且从 1, 2, 3, 4, 5 随机的给 coflow 设置权重，重复这个实验 20 次，图 8.6 显示了实验结果。看到，Yosemite 相对于 TCP 提高程度为 3.5 (Narrow&Short), 1 (Narrow&Long), 1.2 (Wide&Short), 1.1 (Wide&Long) and 1.1 (ALL)，LP-based 的方法的提高幅度为 0.9 (Narrow&Long), 1.3 (Wide&Short), 1.3 (Wide&Long) and 1.2 (ALL)。发现，相对于 LP-based 的方法，Yosemite 存在平均 10% 的性能损失。

#### 8.4.6 在线策略和离线策略的差距

表 8.3 三个策略的对比

| 策略             | 方式 | 复杂性 | 性能 | 是否需要提前预知信息 |
|----------------|----|-----|----|------------|
| 2-approximate  | 离线 | 复杂  | 高  | 需要         |
| simple-offline | 离线 | 简单  | 高  | 需要         |
| Yosemite       | 在线 | 简单  | 高  | 不需要        |

事实上，在 Yosemite 策略中使用了两个放缩来简化策略，简单的离线策略假设

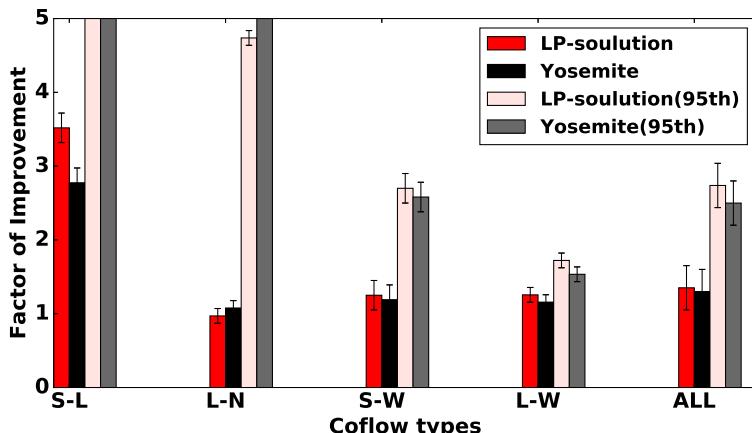
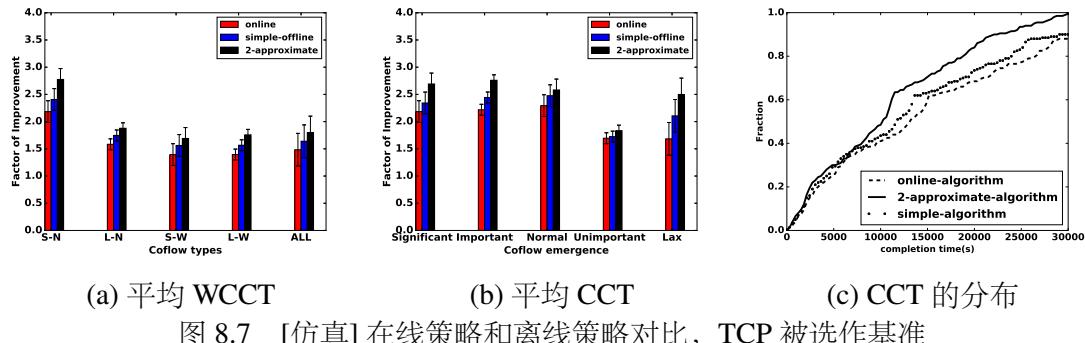


图 8.6 [仿真] LP-based 的策略和 Yosemite 策略性能对比



数据中心中在每个端口上已经进行了负载均衡。在线算法进一步假设长的 coflow 因为持续的时间长，因该有较低的优先级。这两个放缩会损伤策略的性能。为了展示策略性能的损失，假设在 60 台机器的小型数据中心有 100 条 coflow，这些 coflow 同时在  $t=0$  启动，并且设置权重在 1, 2, 3, 4, 5 中随机选取。重复每组实验 100 次，图8.7显示了实验结果。注意，在实验中，error bar 包含平均，最大和最小值。从图8.7(a) 看到，对于在线策略，WCCT 相对于 TCP 提高倍数是  $1.6\times$ ，简单离线策略是  $1.7\times$ ，2-近似策略是  $1.8\times$ 。和 2-近似策略相比，对于优化平均 WCCT，在线策略大约有 10% 的性能损失。图8.7 (b) 显示的是不同紧急程度的 coflow 性能对比。看到在线策略，对于紧急和重要的 coflow，平均提高幅度是  $2\times$  和  $2.1\times$ ，简单离线策略提高幅度是  $2.2\times$  和  $2.3\times$ 。2-近似策略提高幅度是  $2.5\times$ ,  $2.6\times$ 。这启示在线策略大约有 30% 的性能损失。图 8.7(c) 展示了 CCT 的分布情形。发现对于 2-近似策略有超过 80% 的 CCT 在 20000s 以内，对于简单离线策略和在线策略是 70% 和 60%。表8.3展示的是不同策略的对比情形。可以发现，在线策略有一些性能损失，但是在线策略，不需要预先得知流的长度。因为在真实的环境中，一些应用的流的信息是无法预先得知的，因此，不需要预先知道流信息的在线策略应用范围更广。

## 8.5 本章小结

本章中提出了基于重要性和网络拥塞的任务传输调度方案 – Yosemite，在不用预先得知流大小的前提下对 coflow 进行调度，测量并分析 AT&T 的流量，并把 AT&T 的应用根据优先级进行区分。最后，使用 Facebook 和 AT&T 的流量对 Yosemite 性能进行评估。

# 第9章 数据中心传输系统和性能评估

## 9.1 概述

本章首先对数据中心传输系统 FlyTransfer 进行介绍，首先本章对 FlyTransfer 系统架构进行介绍。随后，本章对 FlyTransfer 系统各个组件进行介绍。最后，使用真实流量对 FlyTransfer 系统进行性能评估。

## 9.2 FlyTransfer 系统

本部分我们介绍，数据中心应用传输系统-FlyTransfer 的设计。首先，本章节介绍 FlyTransfer 整体架构。随后，从系统组件，API 等方面对系统的主要实现细节进行介绍。

### 9.2.1 FlyTransfer 架构

FlyTransfer 系统包含三个组件，负责进行任务调度的 master 节点，负责进行任务发送和接收的 worker 节点，负责进行信息备份的 backup 节点。图9.1显示的是 FlyTransfer 系统的架构图。其中 master 节点包括三个部分：和用户进行交互的

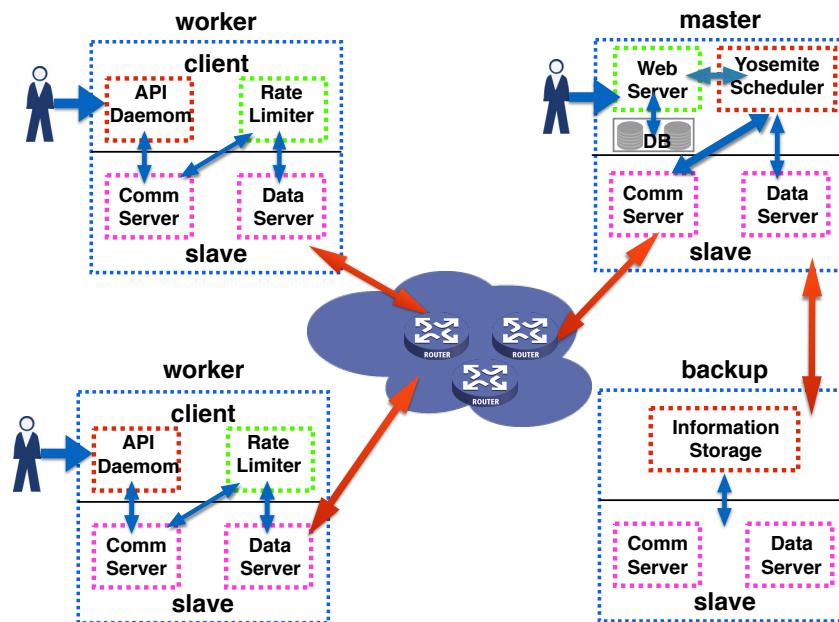


图 9.1 FlyTransfer 系统架构

UI 部分 – web server, 进行 coflow 调度的 scheduler, 调度器可以运行各种 coflow 调度算法。进行底层通信的 slave。worker 节点包含两个部分: 给用户和应用提供 API 的 client 组件, 底层通信 slave 组件。backup 节点是进行备份, 其中 backup 节点包括 information storage 和 slave 两个组件。

## 9.2.2 FlyTransfer 组件

### 9.2.2.1 master 组件

master 组件是 FlyTransfer 系统的“大脑”部分, 是整个系统的核心部分。Fly-Transfer 系统采用集中式控制机制, Master 节点是系统的控制部分。master 节点包括三个部分: 和用户进行交互的 UI 部分 – web server, 进行 coflow 调度的 scheduler, 调度器可以运行各种 coflow 调度算法。用户可以通过 Web Server 查看 coflow 的调度实时状态, Web Server 开放默认端口 16016, 其中主界面显示如图9.2所示。

从图9.2可以看到, 主界面主要有三部分: Slave 的数目, 正在运行 coflow 的信息, 已经完成传输的 coflow 信息。Web Server 是基于 jetty<sup>①</sup>实现的。Web Server 同时把 coflow 调度的信息存储在数据库中。

图9.3(a) 显示的是 Slave 的信息, 其中包括 Slave 标记 (ID) 和 Slave 的发送和接收的信息等。图9.3(b) 显示的是 coflow 的信息, 其中包括 coflow 的 ID 和 coflow

| Yosemite Master at Yosemite://172.17.0.2:1606 |                     |                     |      |          |          |
|-----------------------------------------------|---------------------|---------------------|------|----------|----------|
| URL:Yosemite://172.17.0.2:1606                |                     |                     |      |          |          |
| Slaves:11                                     |                     |                     |      |          |          |
| Coflows:0 Running, 10 Completed               |                     |                     |      |          |          |
| <b>Slaves</b>                                 |                     |                     |      |          |          |
| ID                                            | Address             | State               |      |          |          |
| slave-20180224140513-172.17.0.2-39110         | 172.17.0.2 : 39110  | ALIVE               |      |          |          |
| slave-20180224140513-172.17.0.3-35956         | 172.17.0.3 : 35956  | DEAD                |      |          |          |
| slave-20180224140515-172.17.0.4-41776         | 172.17.0.4 : 41776  | DEAD                |      |          |          |
| slave-20180224140516-172.17.0.5-43932         | 172.17.0.5 : 43932  | DEAD                |      |          |          |
| slave-20180224140518-172.17.0.6-38516         | 172.17.0.6 : 38516  | DEAD                |      |          |          |
| slave-20180224140520-172.17.0.7-44639         | 172.17.0.7 : 44639  | DEAD                |      |          |          |
| slave-20180224140522-172.17.0.8-44373         | 172.17.0.8 : 44373  | DEAD                |      |          |          |
| slave-20180224140523-172.17.0.9-41908         | 172.17.0.9 : 41908  | DEAD                |      |          |          |
| slave-20180224140525-172.17.0.10-33027        | 172.17.0.10 : 33027 | DEAD                |      |          |          |
| slave-20180224140527-172.17.0.11-36790        | 172.17.0.11 : 36790 | DEAD                |      |          |          |
| slave-20180224140528-172.17.0.12-34070        | 172.17.0.12 : 34070 | DEAD                |      |          |          |
| <b>Running Coflows</b>                        |                     |                     |      |          |          |
| ID                                            | Name                | Submitted Time      | User | State    | Duration |
|                                               |                     |                     |      |          |          |
| <b>Completed Coflows</b>                      |                     |                     |      |          |          |
| ID                                            | Name                | Submitted Time      | User | State    | Duration |
| COFLOW-000008                                 | Broadcast-coflow-8  | 2018/02/24 14:05:38 | root | FINISHED | 15 s     |
| COFLOW-000007                                 | Broadcast-coflow-7  | 2018/02/24 14:05:37 | root | FINISHED | 15 s     |

图 9.2 Master 的 UI 主界面

<sup>①</sup> <http://www.eclipse.org/jetty/download.html>



图 9.3 UI 部分：Slave 和 Coflow

的提交时间以及 coflow 当前的状态（是否完成，是否在传输）等。

FlyTransfer 的 Scheduler 是数据中心应用数据流和任务的调度器，在前面章节中介绍的调度策略，coflow 的排序均实现在这个组件中。本组件收集 coflow 的信息，并且计算每条 coflow 中数据流的带宽。计算完毕后，把每条流信息传递给 comm server，然后发放给对应的 worker 节点，相应的 worker 节点按照计算的速率发送数据流。

### 9.2.2.2 Slave 组件

Slave 负责底层通信和数据传输。Slave 包括两个部分，其中 Comm Server 负责信息传递，Data Server 负责数据传输。Comm Server 的信息传递是基于 Akka<sup>①</sup> (Akka 是 JAVA 虚拟机 JVM 平台上构建高并发、分布式和容错应用的工具包。Akka 用 Scala 语言写成，同时提供了 Scala 和 JAVA 的开发接口)。使用 Akka，可以有效增加系统消息的并发性能，提高系统的鲁棒性。Comm Server 进行的信息传递如表9.1所示。其中消息类型可以分成三类，组件的处理，coflow 的处理，数据流的处理。DataServer 主要进行数据的传输，DataServer 可以按照计算而得的速率进行传输。

### 9.2.2.3 worker 组件

Worker 节点进行数据的发送和接收，worker 节点包含两个部分：给用户和应用提供 API 的 client 组件，底层通信 slave 组件，其中 Slave 组件和 Master 部分相同。Worker 节点提供的 API 如表9.2所示。表9.2中 API 中，registerCoflow 是注册 coflow，其中参数为 CoflowDescription，CoflowDescription 是对 coflow 的描述，包含

<sup>①</sup> <http://akka.io>

表 9.1 Comm Server 进行的信息传递

| 消息名称                 | 传输方向              | 说明                      |
|----------------------|-------------------|-------------------------|
| RegisterSlave        | 从 Worker 到 Master | 注册 Slave                |
| RegisteredSlave      | 从 Master 到 Worker | 告知 Worker, Slave 注册成功   |
| RegisterSlaveFailed  | 从 Master 到 Worker | 告知 Worker, Slave 注册失败   |
| Heartbeat            | 从 Worker 到 Master | 心跳信息, 判断 Slave 是否 alive |
| RegisterClient       | 从 Worker 到 Master | 注册 Client               |
| RegisteredClient     | 从 Master 到 Worker | 告知 Worker, Client 注册成功  |
| RegisterCoflow       | 从 Worker 到 Master | 注册 coflow               |
| RegisteredCoflow     | 从 Master 到 Worker | 注册 coflow 成功            |
| RegisterCoflowFailed | 从 Master 到 Worker | 注册 coflow 失败            |
| UnregisteredCoflow   | 从 Master 到 Worker | coflow 传输完毕             |
| AddFlow              | 从 Worker 到 Master | 增加数据流                   |
| AddFlow              | 从 Worker 到 Master | 增加一条数据流                 |
| GetFlow              | 从 Master 到 Worker | 获取数据流信息                 |
| DeleteFlow           | 从 Worker 到 Master | 删除一条数据流                 |

coflow 重要性, coflow 的宽度等信息。unregisterCoflow 是注销 coflow, 当 coflow 结束传输时, 此函数被调用。handlePut 参数是 coflowId 和 flowDescription, 是给系统注册一条属于 coflowId 的数据流。handleGet 是获得当前 coflow 调度的实时信息。handleFlow 是决定是否开启流级别优化, 默认情况下不开启。ChooseScheduler 用来选择调度策略, 默认的调度策略是 Yosemite 算法。

表 9.2 API 信息

| API 名称           | 参数                       | 返回值           | 说明                         |
|------------------|--------------------------|---------------|----------------------------|
| registerCoflow   | CoflowDescription        | CoflowId      | 注册 coflow, 并返回 coflow 的 Id |
| unregisterCoflow | coflowId                 | boolean       | 注销 coflow                  |
| handlePut        | coflowId,flowDescription | boolean       | 加入一条数据流                    |
| handleGet        | coflowId                 | coflowContent | 得到一条 coflow 信息             |
| handleFlow       | boolean,deadline         | boolean       | 是否开启流级别优化, 默认不开启           |
| ChooseScheduler  | ScheduerId               | boolean       | 选择要使用的调度器                  |

worker 节点上的应用程序通过调用 API 来进行传输优化。RateLimiter 主要和

Data Server 协作，接收来自 Master 计算的速率，并且按照这个计算值同 Data Server 共同进行速率的控制。

### 9.2.3 backup 组件

backup 组件是备份节点，主要进行信息备份，其中 master 组件不断的把信息传递给 backup 组件，backup 组件备份正在调度的 coflow 的信息，同时对 master 组件进行监控当 master 出现宕机时，backup 节点会立刻重启 master 组件，并且让 master 组件恢复到宕机前状态。

## 9.3 性能评估

本部分，在私有 openstack 平台上部署 FlyTransfer 系统，并对之进行性能评估。在 openstack 平台同时启动 80 台虚拟机（2 核，4GB 内存）。每台虚拟机安装 Ubuntu16.04 操作系统。在 Traffic Control 模块<sup>[74]</sup>的帮助下，限制每台 VM 网卡的带宽在 1GB/s。首先，我们进行任务级别的性能展示；随后，是流级别的性能评估。最后，我们对系统的开销进行评估。

### 9.3.1 任务级优化对比

选择 FlyTransfer 任务调度策略是 Yosemite，并在数据中心中部署 5 个应用：map-reduce, file-copy, file-distribute, data-backup, data-distribute。这 5 个应用的优先级分别是紧急，重要，正常，不重要，松散。图9.4 (a) 是整体的平均 CCT 实验结果，从实验结果看，使用 FlyTransfer 使用 Yosemite 策略下的 map-reduce, file-copy, file-distribute, data-backup, data-distribute，5 类应用的平均 CCT 分别为：2, 7, 4, 10, 12。使用 Varys 下的 map-reduce, file-copy, file-distribute, data-backup, data-distribute，5 类应用的平均 CCT 分别为：3, 14, 3, 9, 11。使用 TCP，这 5 类应用的平均 CCT 分别为：7, 12, 6, 22, 25。为了避免某些极端值对实验结论的影响，我们绘制了 95th 比例 coflow 的平均 CCT 实验结果，如图9.4 (b) 所示。我们发现使用 FlyTransfer 使用 Yosemite 策略下的 map-reduce, file-copy, file-distribute, data-backup, data-distribute，5 类应用的平均 CCT 分别为 1.9, 6, 3, 9.4, 13。使用 Varys 结果是：3.1, 13, 2.4, 9.2, 10。使用 TCP，这 5 类应用的平均 CCT 分别为：6.8, 11.2, 5, 21, 23。我们看到对于重要的应用 map-reduce, file-copy，使用 FlyTransfer 使用 Yosemite 策略下的性能比 Varys 提高 20% ~ 30%，但是对于不重要的应用，使用 FlyTransfer 使用 Yosemite 策略下的性能比 Varys 性能差 20%。图9.4 (c) 展示的是 file-distribution 的结果，给应用的 coflow 设置 5 个优先级：紧急，重

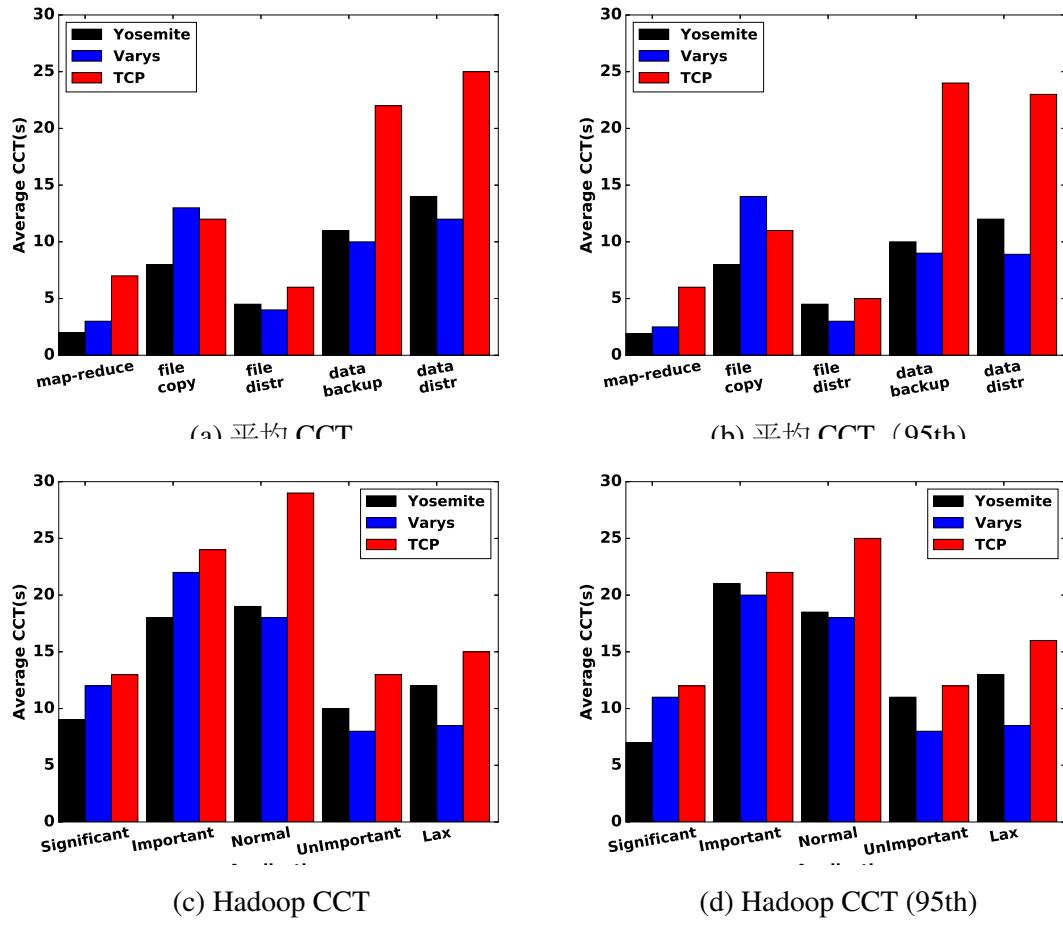


图 9.4 [真实测试] openstack 真实环境下不同应用的测试

要, 正常, 不重要, 松散, 我们发现 Yosemite 性能比 Varys 提高 20%, 图9.4(d)的结果类似。

### 9.3.2 流级优化对比

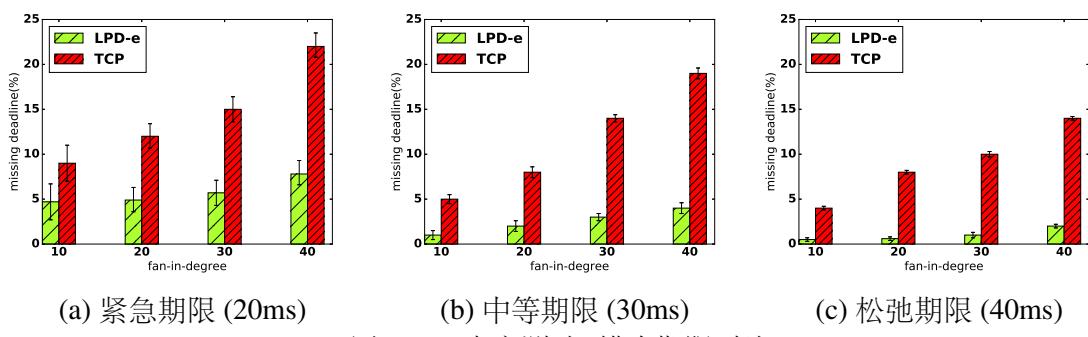


图 9.5 [真实测试] 错失期限对比

图9.5展示错失期限对比。开启对流优化的 API，在 openstack 中构建 OLDI 场景，数据流发送服从柏松分布，假设期限服从指数分布。流的期限平均设置为紧急

期限 (20ms), 中等期限 (30ms) 和松弛期限 (40ms)。从图9.5 (a) 可以看到, 当紧急期限为 20ms 时, 当 fan-in-degree 从 10 到 40 时, LPD-e 下错失期限的比例分别为 5%, 6%, 7%, 8%。而使用 TCP 错失期限的比例分别为 9%, 12%, 14%, 22%。LPD-e 的性能比 TCP 平均提高 2.5×。对于中等期限 (30ms) 和松弛期限 (40ms), 从图9.5 (b) 和从图9.5 (c) 可以看到, 结果和紧急期限 (20ms) 时类似。

### 9.3.3 系统开销

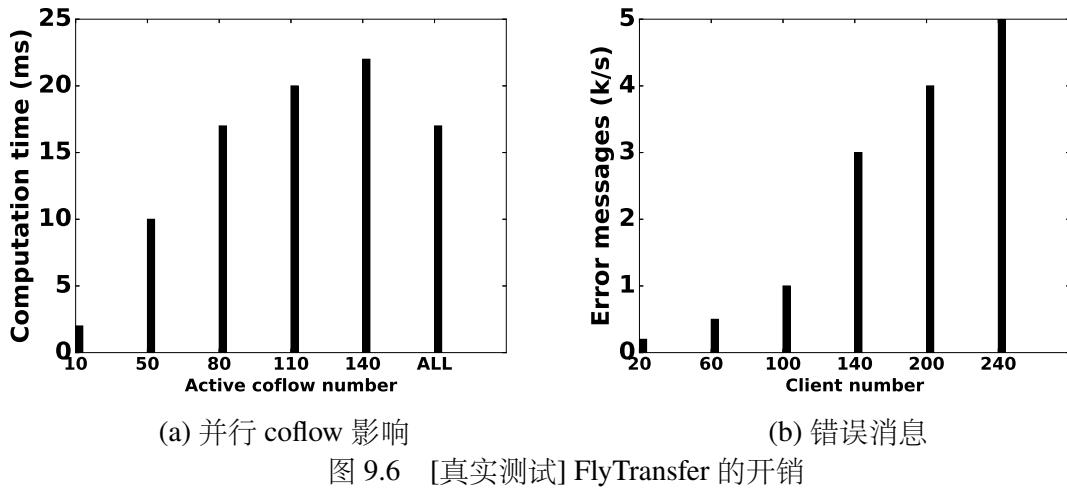


图 9.6 [真实测试] FlyTransfer 的开销

图9.6表示的是 FlyTransfer 的系统开销。在实际中, 随着并行任务和数据流的增多, 控制器的计算时间增大。随着 client 组件增加, 产生出错误消息会增加。图9.6 (a) 展示的是随着并行任务的增加, 控制器的计算时间, 可以看到, 当并行任务在 140 时, 控制器计算时间大约是 22ms, 当并行任务从 10ms 增到 140ms 时, 平均计算时间是 18ms。图9.6 (b) 展示的是随着 client 组件增加, 并行任务的错误消息情形。可以看到, 当 client 数目在 240 时, 错误消息比例是 5k/s, 对比 Varys 系统, 错误消息为 6k/s, 因此, 可以看到 Yosemite 因为错误消息造成的开销比 Varys 小。

## 9.4 本章小结

本章对数据中心传输系统 Fly-Transfer 进行了介绍, 首先, 本章对 Fly-Transfer 整体架构进行了介绍, 然后本章介绍了 Fly-Transfer 的各个组件, 最后, 本章从任务调度, 流传输以及系统开销方面对 Fly-Transfer 进行了性能评估。

## 第 10 章 总结和展望

### 10.1 论文总结

当前越来越多的应用部署在数据中心，不同的应用对数据中心的需求不同，数据中心传统策略 TCP 不能满足不同应用对网络资源的需求。基于此，本文从流级别和任务级别对数据中心的传输进行优化。本文建立了基于 ECN 标记的流传输模型，通过此模型，可以 DCTCP 等策略建模，并分析策略参数；提出了期限敏感流的负载自适应传输优化方案，即使在网络拥塞程度很严重的情形下依然可以根据应用的期限和网络拥塞程度对传输进行优化；提出了基于流持续时间的传输优化方案，在无需得知流大小的前提下，可以同时优化有期限的数据流和流完成时间。提出了数据中心任务级传输优化模型，根据任务级传输优化模型，提出了数据中心传输任务调度的 2 - 近似算法，并证明其近似度。提出了基于流信息的任务级传输优化方案，在预先得知流信息的前提下，对传输任务进行优化。提出了基于重要性和网络拥塞的任务传输调度方案，可以根据应用的重要性和网络拥塞程度对传输任务进行调度。最后，设计并且实现了数据中心传输系统 FlyTransfer，并对之进行性能评估。

本文的主要结论如下：

#### 1 提出了基于 ECN 标记的流传输模型，对基于 ECN 标记的流传输方案进行建模

本文提出了基于 ECN 标记的流传输模型。在基于 ECN 标记的流传输策略中，当网络出现拥塞时，交换机会给数据包标记 CE，当接收端收到被标记的数据包时，对此数据包回复的 ACK 会标记 ECN。发送端根据被标记的 ECN 比例进行拥塞窗口计算。使用基于 ECN 标记的流传输模型，可以对基于 ECN 标记的流传输模型进行建模分析。

#### 2 提出了期限敏感流的负载自适应传输优化方案，当网络拥塞严重时依然能根据期限和网络情形进行速率控制

本文提出了在设计数据流传输方案时，应该遵循一个简单的原则：拥有不同截止期限（deadline）的数据流在带宽分配和占用上应该被区分开，网络负载越重，数据流越应该被区分。根据这个原则，本文提出了一种简单的拥塞控制算法 – 正比负载差分策略（Load Proportional Differentiation，简称 LPD）作为其应用。并在不同的拓扑和负载情况下评估了 LPD。

#### 3 提出了基于流持续时间的传输优化方案，同时可以优化有期限的流以及流

## 完成时间

本文主张使引入流持续时间到拥塞窗调整的过程中。基于此, 本文提出流持续时间速率控制机制 (Flow Duration Time Rate Control, 简称 FDRC)。在不用预先得知流信息的情形下, FDRC 可以减少流错失期限的比例并且能够减小流平均完成时间。本文从理论上分析了 FDRC 的行为, 并在 ns-2 和 Linux 内核实现 FDRC, 并对之进行评估。

## 4 提出了数据中心任务级传输优化模型, 并提出数据中心流组调度的离线调度策略

本文对数据中心任务传输模型进行介绍, 并且从数据中心非阻塞模型, 以及传输任务等对数据中心任务级传输的模型进行介绍。本文推算出数据中心任务调度问题的复杂度, 引入任务的离线调度算法, 并证明离线调度算法的近似度。

## 5 提出了基于流信息的任务级传输优化方案, 可以在预习得知流大小前提下优化任务传输

本文提出基于流信息的任务级传输优化方法来最小化文件平均访问时间 (File Access Time, 简称 FAT)。本文同时结合了最小负载优先的启发式算法做为从 N 个数据块中选取 K 个数据块的方法。在此基础上, 本文设计并实现了 D-Target, 一个集中式调度器, 对文件系统的源选择以及文件系统的任务传输进行整体优化。

## 6 提出了基于重要性和网络拥塞的任务传输调度方案, 可以无需预先得知流大小即能优化任务传输时间

本文引入加权流组完成时间 (Weighted Coflow Completion Time, 简称 WCCT) 问题, 设计一个不需要知道流组 (coflow) 信息就可以进行有效调度的算法, 根据权重和网络拥塞程度调度流组 (coflow), 同时设计一个名为 Yosemite 的调度算法, 并通过数据中心的真实流量对 Yosemite 进行仿真评测。各方案对比, 如表10.1所示。

表 10.1 流级别优化方案

| 策略       | 粒度   | 方法  | 期限 | 任务完成时间 | 预先得知流大小 | 网络拥塞和优化目标关系 |
|----------|------|-----|----|--------|---------|-------------|
| LPD      | 流级别  | 分布式 | 是  | 否      | 是       | 是           |
| FDRC     | 流级别  | 分布式 | 是  | 是      | 否       | 是           |
| D-Target | 任务级别 | 集中式 | 否  | 是      | 是       | 是           |
| Yosemite | 任务级别 | 集中式 | 否  | 是      | 否       | 是           |

## 7 设计并且实现了数据中心传输系统, 实现了数据中心应用流和任务的传输

## 优化

本文设计并实现了数据中心传输系统 FlyTransfer，一个基于集中式的调度系统，FlyTransfer 可以对数据中心的应用实现流级别和任务级别的传输优化和调度。最后，本文将 FlyTransfer 部署在 openstack 平台上，并对之进行性能评估。

## 10.2 进一步研究工作

进一步研究工作包括以下今个方面

### 1 虚拟环境下流传输优化

最近，随着 SDN 等技术的兴起，虚拟化网络技术用途越来越广泛，许多应用开始部署在虚拟环境中。最近的一些研究<sup>[75,76]</sup> 在虚拟化环境从传输层进行优化。VCC<sup>[76]</sup>，在 hypervisor 上进行优化，解决了虚拟化和物理环境下基于 ECN 标记的方法和非 ECN 标记方法的不公平问题。AD/DC<sup>[75]</sup> 在虚拟环境下实现 DCTCP。然后，到目前为止，还没有对虚拟环境下基于期限进行传输优化的方法，因此，在虚拟环境下根据网络拥塞情形以及 CPU 使用率，应用特性进行调度是未来研究的热点问题。

### 2 虚拟环境下任务传输优化

随着 Docker 等容器技术的发展和成型，诸如 Hadoop，Spark 等分布式应用开始部署在容器上。通物理传输相同，仅仅流级别传输是不够的，需要进行任务级的传输优化。仅仅将物理平台上的传输方案平移到虚拟平台上是不够的，因为许多容器共享物理机的 CPU，这会导致传输震荡大。因此，需要结合任务放置或者 CPU 使用率方面进行传输优化。当前，还未出现虚拟环境下任务传输优化系统和平台。

### 3 分布式任务传输

无论是 Varys 还是 Aalo，都需要借助集中处理器进行或多或少的计算。集中处理的方式可以根据网络和应用的整体情况进行调度。集中式的方法计算准确，但是存在两个不足。首先，集中式方式会增加延迟。采取集中控制的方法，节点需要将信息发送给中央控制器，然后中央控制器进行计算，随后把结果反馈给节点，这个过程会增大延迟。其次，集中控制的方式，控制器容易成为系统瓶颈。集中控制的方法，中央控制器存储着网络所有节点的信息，如果中央控制器发生故障或者瘫痪，会使网络整体进入瘫痪状态。因此，分布式的任务传输平台在鲁棒性以及性能上会更有优势。当前业界还没有出现类似解决方案。

### 4 结合路径选择的任务传输优化

当前数据中心对于任务的传输优化，大多假设数据中心是非阻塞结构，在此

结构下，拥塞只发生在数据传输的第一跳和最后一跳。事实上，在数据中心，拥塞不仅发生在第一跳和最后一跳，中间路径也会发生拥塞。因此，拓展非阻塞模型对任务进行传输优化是未来研究的重点内容。

## 参考文献

- [1] 房秉毅, 张云勇, 程莹, 等. 云计算国内外发展现状分析[J]. 电信科学, 2010(S1): 1–6.
- [2] 吴小芳. 数据中心发展趋势探讨[J]. 中国新通信, 2017, 19(1): 19–21.
- [3] 魏祥麟, 陈鸣, 范建华, 等. 数据中心网络的体系结构[J]. 软件学报, 2013(2): 295–316.
- [4] Latency. Latency is everywhere and it costs you sales - how to crush it[EB/OL]. 2009. <http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it>.
- [5] Dean J, Ghemawat S. Simplified data processing on large clusters[J]. In Proceedings of Operating Systems Design and Implementation (OSDI, 2004, 51(1): 107–113.
- [6] Chowdhury M, Zaharia M, Ma J, et al. Managing data transfers in computer clusters with orchestra[J]. Acm Sigcomm Computer Communication Review, 2011, 41(4): 98–109.
- [7] Olston C, Reed B, Srivastava U, et al. Pig latin:a not-so-foreign language for data processing[C]// ACM SIGMOD International Conference on Management of Data. 2008: 1099–1110.
- [8] Isard M, Budiu M, Yu Y, et al. Dryad:distributed data-parallel programs from sequential building blocks[J]. ACM SIGOPS Operating Systems Review, 2007, 41(3): 59–72.
- [9] Yu Y, Isard M, Fetterly D, et al. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language[C]//Usenix Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, Usa, Proceedings. 2008: 1–14.
- [10] Chaiken R, Jenkins B, Ramsey B, et al. Scope: easy and efficient parallel processing of massive data sets[J]. Proceedings of the Vldb Endowment, 2008, 1(2): 1265–1276.
- [11] Chambers C, Raniwala A, Perry F, et al. Flumejava: easy, efficient data-parallel pipelines[J]. Acm Sigplan Notices, 2010, 45(6): 363–375.
- [12] Guo Z, Fan X, Chen R, et al. Spotting code optimizations in data-parallel pipelines through periscope[J]. IEEE Transactions on Parallel and Distributed Systems, 2015, 26(6): 1718–1731.
- [13] Zhang J, Zhou H, Chen R, et al. Optimizing data shuffling in data-parallel computation by understanding user-defined functions[J]. Functions Jiaxing, 2012: 22–22.
- [14] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing[C]//Usenix Conference on Networked Systems Design and Implementation. 2012: 2–2.
- [15] Malewicz G, Austern M H, Bik A J C, et al. Pregel: a system for large-scale graph processing [C]//ACM SIGMOD International Conference on Management of Data. 2010: 135–146.
- [16] Alizadeh M, Greenberg A, Maltz D A, et al. Data center tcp (dctcp)[C]//2010: 63–74.
- [17] Kohavi R, Longbotham R, Sommerfield D, et al. Controlled experiments on the web: survey and practical guide[J]. Data mining and knowledge discovery, 2009, 18(1): 140–181.
- [18] Decandia G, Hastorun D, Jampani M, et al. Dynamo: amazon’s highly available key-value store [C]//ACM Sigops Symposium on Operating Systems Principles. 2007: 205–220.
- [19] Renesse R V, Birman K P, Dan D, et al. Scalable management and data mining using astrolabe[C]// Revised Papers from the First International Workshop on Peer-to-Peer Systems. 2002: 280–294.

- [20] Vasudevan V, Phanishayee A, Shah H, et al. Safe and effective fine-grained tcp retransmissions for datacenter communication[J]. *Sigcomm Comput.commun.rev*, 2009, 39(4): 303–314.
- [21] Phanishayee A, Krevat E, Vasudevan V, et al. Measurement and analysis of tcp throughput collapse in cluster-based storage systems[C]//2008: 12:1–12:14.
- [22] Chen Y, Griffith R, Liu J, et al. Understanding tcp incast throughput collapse in datacenter networks[C]//2009: 73–82.
- [23] Vamanan B, Hasan J, Vijaykumar T. Deadline-aware datacenter tcp (d2tcp)[C]//2012: 115–126.
- [24] Munir A, Qazi I, Uzmi Z, et al. Minimizing flow completion times in data centers[C]//INFOCOM, 2013 Proceedings IEEE. 2013: 2157–2165.
- [25] Mittal R, Dukkipati N, Blem E, et al. Timely: Rtt-based congestion control for the datacenter [C]//ACM SIGCOMM Computer Communication Review: volume 45. : ACM, 2015: 537–550.
- [26] Wu H, Feng Z, Guo C, et al. Ictcp: Incast congestion control for tcp in data-center networks[J]. *Networking, IEEE/ACM Transactions on*, 2013, 21(2): 345–358.
- [27] Wilson C, Ballani H, Karagiannis T, et al. Better never than late: Meeting deadlines in datacenter networks[C]//2011: 50–61.
- [28] Hong C Y, Caesar M, Godfrey P. Finishing flows quickly with preemptive scheduling[J]. *ACM SIGCOMM Computer Communication Review*, 2012, 42(4): 127–138.
- [29] Bai W, Chen K, Wang H, et al. Information-agnostic flow scheduling for commodity data centers. [C]//NSDI. 2015: 455–468.
- [30] Grosvenor M P, Schwarzkopf M, Gog I, et al. Queues don't matter when you can jump them! [C]//NSDI. 2015: 1–14.
- [31] Perry J, Ousterhout A, Balakrishnan H, et al. Fastpass: A centralized zero-queue datacenter network[J]. *ACM SIGCOMM Computer Communication Review*, 2015, 44(4): 307–318.
- [32] Alizadeh M, Yang S, Sharif M, et al. pfabric: Minimal near-optimal datacenter transport[C]// 2013: 435–446.
- [33] Zats D, Das T, Mohan P, et al. Detail: Reducing the flow completion time tail in datacenter networks[C]//2012: 139–150.
- [34] Chen L, Chen K, Bai W, et al. Scheduling mix-flows in commodity datacenters with karuna [C]//Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference. : ACM, 2016: 174–187.
- [35] Munir A, Baig G, Irteza S M, et al. Friends, not foes: synthesizing existing transport strategies for data center networks[C]//Proceedings of the 2014 ACM conference on SIGCOMM. : ACM, 2014: 491–502.
- [36] Dogar F R, Karagiannis T, Ballani H, et al. Decentralized task-aware scheduling for data center networks[C]//ACM SIGCOMM Computer Communication Review: volume 44. : ACM, 2014: 431–442.
- [37] Chowdhury M, Zhong Y, Stoica I. Efficient coflow scheduling with varys[C]//ACM SIGCOMM Computer Communication Review: volume 44. : ACM, 2014: 443–454.
- [38] Chowdhury M, Stoica I. Coflow: A networking abstraction for cluster applications[C]// Proceedings of the 11th ACM Workshop on Hot Topics in Networks. : ACM, 2012: 31–36.

- [39] Chowdhury M, Stoica I. Efficient coflow scheduling without prior knowledge[C]//ACM SIGCOMM Computer Communication Review: volume 45. : ACM, 2015: 393–406.
- [40] Zhang H, Chen L, Yi B, et al. Coda: Toward automatically identifying and scheduling coflows in the dark[C]//Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference. : ACM, 2016: 160–173.
- [41] Luo S, Yu H, Zhao Y, et al. Towards practical and near-optimal coflow scheduling for data center networks[J]. IEEE Transactions on Parallel and Distributed Systems, 2016, 27(11): 3366–3380.
- [42] Roemer T A. A note on the complexity of the concurrent open shop problem[J]. Journal of scheduling, 2006, 9(4): 389–396.
- [43] Zhao Y, Chen K, Bai W, et al. Rapier: Integrating routing and scheduling for coflow-aware data center networks[C]//Computer Communications (INFOCOM), 2015 IEEE Conference on. : IEEE, 2015: 424–432.
- [44] Liu S, Huang J, Zhou Y, et al. Task-aware tcp in data center networks[C]//Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on. : IEEE, 2017: 1356–1366.
- [45] Qiu Z, Stein C, Zhong Y. Minimizing the total weighted completion time of coflows in data-center networks[C]//Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures. : ACM, 2015: 294–303.
- [46] Zhang H, Shi X, Yin X, et al. Fdrc-flow duration time based rate control in data center networks [C]//Quality of Service (IWQoS), 2016 IEEE/ACM 24th International Symposium on. : IEEE, 2016: 1–10.
- [47] Meisner D, Sadler C M, Barroso L A, et al. Power management of online data-intensive services [C]//2011: 319–330.
- [48] The network simulator — ns-2[EB/OL]. <http://www.isi.edu/nsnam/ns/>.
- [49] Simulation code of lpd[EB/OL]. <https://github.com/zhanghan1990/LPD-sim/>.
- [50] Linux kernel code of lpd[EB/OL]. <https://github.com/zhanghan1990/LPD/>.
- [51] Wischik D, Raiciu C, Greenhalgh A, et al. Design, implementation and evaluation of congestion control for multipath tcp.[C]//NSDI: volume 11. 2011: 8–8.
- [52] Alizadeh M, Javanmard A, Prabhakar B. Analysis of dctcp: stability, convergence, and fairness [C]//Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems. : ACM, 2011: 73–84.
- [53] Scalability H. Latency is everywhere and it costs you sales - how to crush it[EB/OL]. <http://simula.stanford.edu/~alizade/Site/DCTCP.html>.
- [54] Lee J, Turner Y, Lee M, et al. Application-driven bandwidth guarantees in datacenters[C]// Proceedings of the 2014 ACM conference on SIGCOMM. : ACM, 2014: 467–478.
- [55] The dummynet project. [online]. available[EB/OL]. <http://info.iet.unipi.it/lugi/dummynet/>.
- [56] Cao Y, Xu M, Fu X, et al. Explicit multipath congestion control for data center networks[C]// Proceedings of the ninth ACM conference on Emerging networking experiments and technologies. : ACM, 2013: 73–84.
- [57] Zhang H, Shi X, Yin X, et al. More load, more differentiation—a design principle for deadline-aware congestion control[C]//Computer Communications (INFOCOM), 2015 IEEE Conference on. : IEEE, 2015: 127–135.

- [58] Bar-Noy A, Halldórsson M M, Kortsarz G, et al. Sum multicoloring of graphs[J]. *Journal of Algorithms*, 2000, 37(2): 422–450.
- [59] Huang X S, Sun X S, Ng T E. Sunflow: Efficient optical circuit scheduling for coflows[C]// Proceedings of the 12th International Conference on emerging Networking EXperiments and Technologies. : ACM, 2016: 297–311.
- [60] Mastrolilli M, Queyranne M, Schulz A S, et al. Minimizing the sum of weighted completion times in a concurrent open shop[J]. *Operations Research Letters*, 2010, 38(5): 390–395.
- [61] Chen Z L, Hall N G. Supply chain scheduling: Assembly systems[R]. : Working Paper, Department of Systems Engineering, University of Pennsylvania, 2000.
- [62] Qiu Z, Stein C, Zhong Y. Experimental analysis of algorithms for coflow scheduling[C]// International Symposium on Experimental Algorithms. 2016: 262–277.
- [63] Kumar A, Manokaran R, Tulsiani M, et al. On lp-based approximability for strict csp[C]// Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete Algorithms. : Society for Industrial and Applied Mathematics, 2011: 1560–1573.
- [64] Sathiamoorthy M, Asteris M, Papailiopoulos D, et al. Xoring elephants: Novel erasure codes for big data[C]//Proceedings of the VLDB Endowment: volume 6. : VLDB Endowment, 2013: 325–336.
- [65] Wu J, Ping L, Ge X, et al. Cloud storage as the infrastructure of cloud computing[C]//Intelligent Computing and Cognitive Informatics (ICICCI), 2010 International Conference on. : IEEE, 2010: 380–383.
- [66] Beaver D, Kumar S, Li H C, et al. Finding a needle in haystack: Facebook’s photo storage.[C]// OSDI: volume 10. 2010: 1–8.
- [67] Li J, Li B. Erasure coding for cloud storage systems: a survey[J]. *Tsinghua Science and Technology*, 2013, 18(3): 259–272.
- [68] Dimakis A G, Prabhakaran V, Ramchandran K. Decentralized erasure codes for distributed networked storage[J]. arXiv preprint cs/0606049.
- [69] Lin H Y, Tzeng W G. A secure erasure code-based cloud storage system with secure data forwarding[J]. *IEEE transactions on parallel and distributed systems*, 2012, 23(6): 995–1003.
- [70] Dimakis A G, Godfrey P B, Wu Y, et al. Network coding for distributed storage systems[J]. *IEEE Transactions on Information Theory*, 2010, 56(9): 4539–4551.
- [71] Schurman E, Brutlag J. The user and business impact of server delays, additional bytes, and http chunking in web search[C]//Velocity Web Performance and Operations Conference. 2009.
- [72] Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters[J]. *Communications of the ACM*, 2008, 51(1): 107–113.
- [73] D-target[EB/OL]. <https://github.com/zhanghan1990/D-Target/>.
- [74] Traffic control[EB/OL]. <http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html/>.
- [75] He K, Rozner E, Agarwal K, et al. Ac/dc tcp: Virtual congestion control enforcement for datacenter networks[C]//Conference on ACM SIGCOMM 2016 Conference. 2016: 244–257.
- [76] Cronkite-Ratcliff B, Bergman A, Keslassy I, et al. Virtualized congestion control[C]//Conference on ACM SIGCOMM 2016 Conference. 2016: 230–243.

## 致 谢

首先感谢单位和清华给了我这次继续深造的机会，让我进入了一个更大的人生舞台，有机会结识了许多值得尊敬的老师和才华横溢的小伙伴们，从此我的博士生涯不再孤单，开始奋勇前行。在清华学习期间，我不仅学到了尖端的科学知识、科学的研究方法，更学到了实事求是、大胆而严谨的工作作风，这将是我一生的宝贵财富。

衷心感谢我的导师尹霞教授，她渊博的知识、开阔的眼界、高效的办事风格、亲切而耐心的指导和交流方式等时刻影响着我，使我受益终身！

非常感谢施新刚老师，施老师学识广博、工作严谨、一丝不苟，具体而细致的指导了我在博士期间的每一项工作，是我的良师益友！

非常感谢王之梁老师，严谨的科研作风深入到了王老师的骨髓里，每一次对我提出问题的解答和细节的纠正，都使我进步很大！

感谢余家傲老师和张娜老师，她们是我们的后勤部长，正是因为她们的辛勤付出才使我们有了轻松的学习环境。感谢姚姜源、吴丹、耿海军，郭迎亚，田庚以及实验室的师弟师妹们在博士期间对我的无私帮助！

最后感谢我的家人对我的体谅和无私付出，让我有了完整的学习和科研时间，保证了博士的顺利毕业。

声 明

---

声 明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签 名： \_\_\_\_\_ 日 期： \_\_\_\_\_

## 个人简历、在学期间发表的学术论文与研究成果

### 个人简历

1990 年 4 月 17 日出生于山东省淄博市。

2009 年 9 月考入吉林大学计算机科学与技术专业，2013 年 7 月本科毕业并获得学士学位。

2013 年 9 月考入清华大学计算机科学与技术系攻读工学博士至今。

2016 年 10 月 – 2017 年 9 月美国乔治华盛顿大学访问学者。

### 发表的学术论文

- [1] **Zhang H**, Shi X, Guo Y, et al. Efficient Scheduling of Weighted Coflows in Data Centers. s, 2017.(SCI 检索,CCF A 类期刊, 审稿中).
- [2] **Zhang H.**, Shi, X., Guo, Y., Wang, Z., & Yin, X. (2017). More load, more differentiation—Let more flows finish before deadline in data center networks. *Computer Networks*, 127, 352-367.7. (SCI 检索,CCF B 类期刊).
- [3] **Zhang H**, Shi, X., Yin, X., & Wang, Z. (2017, October). Joint source selection and transfer optimization for erasure coding storage system. In 2017 IEEE/ACM 36th International Performance Computing and Communications Conference(IPCCC). (pp. 1-10). IEEE.(EI 检索,CCF C 类会议)
- [4] **Zhang H.**, Shi, X., Yin, X., & Wang, Z. (2017, October). Yosemite: Efficient scheduling of weighted coflows in data centers. In 2017 IEEE 25th International Conference on Network Protocols (ICNP)(pp. 1-2). IEEE..(EI 检索,CCF B 类 poster)
- [5] **Zhang H**, Shi, X., Yin, X., Wang, Z., & Guo, Y. (2016, June). FDRC-Flow duration time based rate control in data center networks. In Quality of Service (IWQoS), 2016 IEEE/ACM 24th International Symposium on(pp. 1-10). IEEE..(EI 检索,CCF B 类会议)
- [6] **Zhang H**, Shi, X., Yin, X., Ren, F., & Wang, Z. (2015, April). More load, more differentiation—A design principle for deadline-aware congestion control. In Computer Communications (INFOCOM), 2015 IEEE Conference on(pp. 127-

- 135). IEEE. .(EI 检索,CCF A 类会议)
- [7] Geng, H., Shi, X., Yin, X., Wang, Z., & **Zhang H.** (2015, June). Algebra and algorithms for efficient and correct multipath QoS routing in link state networks. In *Quality of Service (IWQoS), 2015 IEEE 23rd International Symposium on*(pp. 261-266). IEEE. (EI 检索,CCF B 类会议)
- [8] Zhao, Z., Li, Q., Xu, M., Shi, X., & **Zhang H.** (2016, May). Reduce completion time and guarantee throughput by transport with slight congestion. In *Communications (ICC), 2016 IEEE International Conference on*(pp. 1-6). IEEE.(EI 检索,CCF C 类会议)

### 参与的科研项目

- [1] 清华-思科联合实验室项目，“OnePK based Network Traffic Data Collection and Analysis”
- [2] 清华-思科联合实验室项目，“Data Center Traffic Generator”
- [3] 863 项目，“软件定义网络（SDN）验证与示范”