密级：_____

中国科学院大学
University of Chinese Academy of Sciences

硕士学位论文

由应用驱动的数据中心流调度策略

作者姓名：王　壮

指导教师：陈明宇　研究员

中国科学院计算技术研究所

学位类别：工学硕士

学科专业：计算机系统结构

研 究 所：中国科学院计算技术研究所

二〇一七年五月

An Application-driven Flow Scheduling in Data Centers

University of Chinese Academy of Sciences

A Thesis Submitted to

The University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Master of Science

in

Computer Architecture

by

Zhuang Wang

Thesis Supervisor: Professor Mingyu Chen

Institute of Computing Technology

Chinese Academy of Sciences

May, 2017

声　　明

我声明本论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，本论文中不包含其他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

作者签名：　　日期：

论文版权使用授权书

本人授权中国科学院计算技术研究所可以保留并向国家有关部门或机构送交本论文的复印件和电子文档，允许本论文被查阅和借阅，可以将本论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编本论文。

（保密论文在解密后适用本授权书。）

作者签名：　　导师签名：　　日期：

摘　要

保证延时敏感型应用产生的流量的低延时和最小化吞吐密集型应用产生的数据流的平均流完成时间是数据中心内性能优化的两个重要指标。现有的方案通常只优化了其中一个指标，或是需要假设相关数据流信息已知。但在很多应用中，数据流信息很难在数据传输结束前获得。所以，我们的目标是要找到一种不需要假设数据流信息就能同时优化两个指标的数据流调度方案。

我们提出了 Panda，一种不需要假设数据流信息，而是通过分析运行在数据中心内的应用的流量特征来同时优化两个指标的流调度策略。我们分析了 Facebook 数据中心内的流量特征后发现，延时敏感型应用产生的数据包大小偏小，而吞吐密集型应用却倾向于产生较大的数据包。所以，Panda 根据这一特性对两种应用产生的流量进行区分。Panda 的核心思想是，找到一个最优阈值，根据大小将数据包分成大包和小包，保证小包在延时敏感型应用中占主要部分，而大包则主要集中在吞吐密集型应用中；然后为每条数据流分配一个计数器，当数据包到达时，更新计数器的值：如果数据包为大包，增加计数器的值，否则减小计数器的值；之后根据计数器的大小为数据流分配优先级；最后配合多优先级队列，使得延时敏感型应用产生的数据流一直处于高优先级队列，从而保证其低延时，而吞吐密集型应用产生的数据流的优先级则随着其已经发送的数据量的增加而逐渐降低，从而近似实现了短流优先策略，优化了其平均流完成时间。

我们在 Linux 平台上实现了一个 Panda 的原型系统，并评测了其性能。实验结果显示，Panda 能有效保证延时敏感型应用的低延时要求。与 PIAS 比较，当延时敏感型应用产生的数据流大小较大时，Panda 能降低其请求延时的 99% 分位值的 50% 左右，并能同时保证与 PIAS 基本一致的平均流完成时间优化性能。

关键词：数据中心网络；流调度策略；应用驱动

Abstract

Bounding latency for delay-sensitive applications (DSA) and minimizing the flow complete times (FCT) for throughput-intensive applications (TIA) are two main metrics to evaluate the performance of data center networks (DCN). While most of existing approaches just focus on either of them or assume that some flow information, such as flow size or priority, is known *a priori*. However, such information is hard to obtain, or even simply not available in many cases. By contrast, we seek to optimize both metrics with no prior knowledge.

To this end, we present Panda, a DCN flow scheduling scheme to bound low latency for DSA and optimize FCT for TIA on the premise that flow information is not known *a priori*. We observe that the majority of packets generated by DSA are small, while by TIA are large. Therefore, Panda takes advantage of the two distinct flow size distributions to differentiate the two kinds of applications. At its heart, Panda derives an optimal threshold to divide packets into two categories: large and small, ensuring that small packets dominate traffic from DSA and large ones dominate traffic from TIA. In addition, Panda allocates each flow a counter which is initiated with zero. Large packets increase the counter while small packets decrease it. Then Panda assigns priorities to flows according to their counters, achieving the two goals through prioritizing flows from DSA and emulating Shortest Job First for flows from TIA.

We have implemented a Panda prototype and evaluated PIAS through testbed experiments. Our evaluation results show that Panda can effectively bound low latency for DSA. For example, it reduces the query complete time in the 99th percentile by up to 50% over PIAS while maintaining the similar performance on FCT.

Keywords: Data center network; Flow scheduling; Application-driven scheduling

目　录

图目录

表目录

第一章　引言

为了充分利用数据中心的物理资源，如 CPU、内存等，数据中心通常会同时运行多种应用 [1–4]。比如，在 Morgan Stanley 的数据中心内，同时运行着 Key-Value Store 和分布式文件系统等应用 [3]; 在 Microsoft 的数据中心内，同时运行着 MapReduce、分布式文件系统、网页服务等应用 [1,2]。这些应用除了共享硬件资源外，还会共享数据中心的网络资源。不同的应用可能会要求不同的服务质量保证（Quality of Service，QoS），如网页服务会要求低延时，而数据备份应用虽然对延时要求不高，但需要占用较高带宽。因此，数据中心需要为不同的应用同时提供不同的服务质量保证。这些应用可以从服务质量保证的角度大致分成两类：延时敏感型应用和吞吐密集型应用。对于延时敏感型应用，其产生的请求通常会有严格的时间要求限制 [5–8]；而对于吞吐密集型应用，则需要较高的网络带宽对它们产生的流量进行传输。

1.1　数据中心内服务质量保证面临的挑战

由于对服务质量有不同需求的多种应用同时运行在同一个数据中心内，这会导致它们不可避免地竞争网络资源。当不同的应用竞争同一条瓶颈链路时，很难保证所有应用的服务质量都得到满足。如图 1.1 所示，延时敏感型应用 L 和吞吐密集型应用 T 共享一条瓶颈链路。如果没有服务质量保证，系统将会使用相同的策略调度这两种应用产生的流量。由于需要较高带宽传递数据，应用 T 很有可能会在瓶颈链路中形成队列，导致应用 L 在该瓶颈链路处遭受较大的排队延时。

应用区分 数据中心能为应用提供服务质量保证的前提是能准确区分应用。由于多种应用同时在数据中心内运行，它们产生的流量会混合在一起。如果不对应用加以区分，就很难对不同应用产生的流量提供其所需要的服务质量保证。目前应用的区分方案主要可以分成两类, 第一类是应用在其产生的流量上主动标记其应用类型或者所需要的服务质量 [6,9–11]，调度算法可以直接获取该信息; 第二类是应用产生的流量没有被预先区分，调度算法需要主动区分应用类型 [12]。

延时敏感型应用的延时保证 延时敏感型应用，如网页搜索、社交网络、推荐系统等，对网络宽带的需求不大，但其产生的请求通常会有严格的时间限制。如果在该时间限制能请求未能应答，该请求将不会成为返回结果的一部分，从而影响用户体验；由于系统对于该请求仍然会响应，但该响应并不会增加用户体验，因此造成了带宽的浪费，导致服务商的利益受损 [8]。因此，数据中心需要保证延时敏感型应用的低延时。

最小化吞吐密集型应用的平均流完成时间 吞吐密集型应用，如虚拟机迁移、数据备份等，它们对流完成时间虽然没有明确的时间限制，但这些应用还是希望能在更短的

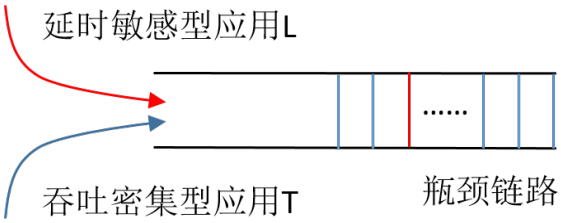


图 1.1　不同服务质量保证需求的应用竞争同一条瓶颈链路.

时间内完成任务。评价调度算法好坏的指标之一是平均流完成时间。平均流完成时间的定义为：数据流从开始等待服务到服务完成所需要的平均时间。

假设有三条数据流 a, b, c 同时请求服务，它们的大小分别为 1,2,4 个单位时间长度。若我们只考虑如表 1.1 所示的 6 种调度策略，可以发现，按照 a, b, c 的顺序对这三条数据流进行调度的平均流完成时间最短。最小化平均流完成时间是一个经典问题，对这个问题的分析可以分成两类：数据流大小已知 [6,11,13] 与数据流大小未知 [10,12,14]。回到刚才调度 a, b, c 三条数据流的例子。如果已知三条流的大小，我们可以根据流大小从小到大的顺序对数据流进行调度，该算法可以保证平均流完成时间最小 [14]。但在很多种应用中，数据流的大小很难甚至无法预知，如在 HTTP 1.1 中支持的分块传输（chunked transfer）[15]，数据库请求响应 [16] 和流处理 [17] 等。在这种情况下，只能采用一些不需要预知数据流大小的调度算法，如 PIAS[12],LAS[14]（Least Attained Service）等。

表 1.1　流大小分别为 1,2,4 的 a,b,c 三条流的部分调度策略

策略编号	调度策略	平均流完成时间
1	a, b, c	$11/3$
2	a, c, b	$13/3$
3	b, a, c	$12/3$
4	b, c, a	$15/3$
5	c, a, b	$16/3$
6	c, b, a	$17/3$

交换机排队延时　当经过交换机的负载流量较大，入口速率大于对应的出口速率时，数据包会在交换机内产生队列，导致排队延时的产生。交换机产生的排队延时在数据包从发送端到接收端所产生的总延时中占有较大比例 [5]。数据中心传统的多层网络如图 1.2 所示。数据流从物理机产生后，会流经接入层、汇聚层、核心层，排队延时在这三部分都可能产生。为了降低数据包在交换机类所经历的排队延时而提出的方案包括 DCTCP[5],pFabric[11],PIAS[12],QJUMP[9],TIMELY[18],Fastpass[19] 等。

终端排队延时　数据包在终端也会经历排队延时。随着虚拟化技术 [20–22] 与容器技术 [23] 的发展，一台物理机可以同时实例化多台虚拟机或者容器。数据从实例产生后在到达物理网卡前所经过的处理模块如图 1.3 所示。流量从实例产生后，会先到达网桥

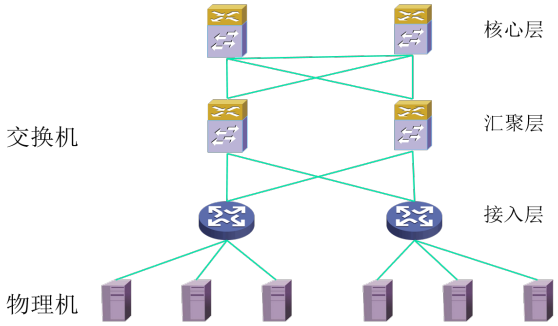


图 1.2 数据中心中的三层网络拓扑

（容器技术为 TCP/IP Stack），再进入 QDisc Queue，进行限速、多优先级调度等操作，之后进入物理网卡的驱动队列。当实例产生的流量较大时，数据包会在 QDisc Queue 内产生队列，导致排队延时的产生。更严重的是，终端排队延时与交换机排队延时是相互独立的两部分，当数据包在终端经历了排队延时后，仍有可能在交换机内再次经历排队延时。

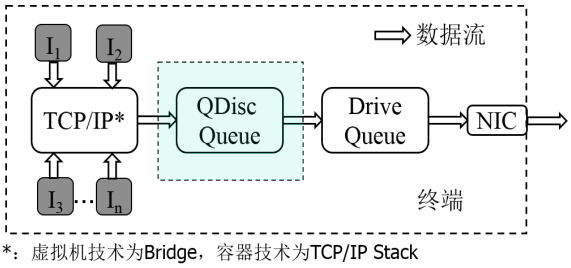
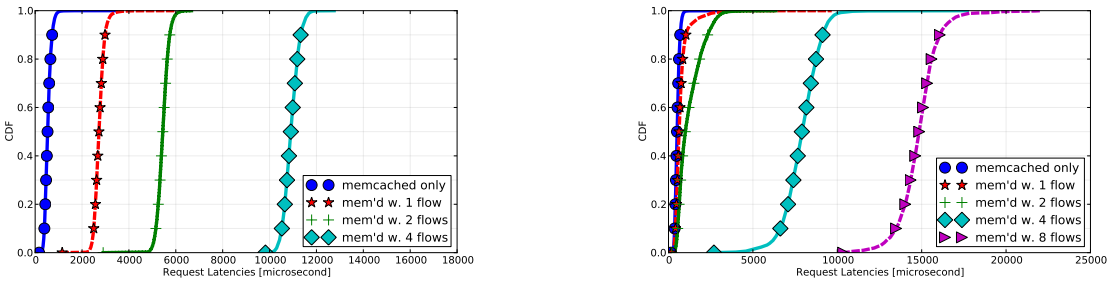


图 1.3 在终端产生的数据流

终端排队延时的严重性 如果缺少有效的拥塞控制算法，数据包在终端经历的排队延时可能比在交换机内经历的排队延时更加严重。为了验证终端排队延时的严重性，我们将两台物理机 PM_1，PM_2 直连（排除交换机排队延时的干扰），如图 1.4 所示。在 PM_1 中，我们同时运行了两种应用：Memcached[24] 和 iPerf[25]。Memcached 是一种在 Facebook 数据中心内广泛使用的 Key-Value Store 数据库，是一种典型的延时敏感型应用。与此同时，Matthew 等人 [9] 修改过的 Memcached 版本还可以测量每个 GET/SET 请求的延时，因此，我们可以使用该应用评估排队延时的严重程度。iPerf 通常用于测试网络的链路带宽，可以认为是一种典型的吞吐密集型应用。实验中，Memcached 应用的客户端运行在 PM_2 上，服务器程序运行在 PM_1 上，GET 请求从 PM_2 流向 PM_1，响应从 PM_1 返回 PM_2。iPerf 应用的客户端运行在 PM_1 上，服务器程序运行在 PM_2 上，数据从 PM_1 流向 PM_2，对 Memcached 的响应数据进行干扰。PM_1 采用的 AQM（Active Queue Management）是 *fifo*，默认可容纳数据包的个数为 1000，拥塞控制算法为 CUBIC[26]。我们分别在 1Gbps 与 10Gbps 网络环境中进行了测试。实验结果如图 1.5 所示。

图 1.5(a) 表示的是在 1Gbps 网络环境下 Memcached 的 GET 请求在不同数量的大流（large flows）干扰下的请求延时。由于在 1Gbps 环境中不启用 offload 特性也能充分利用

图 1.4　测试终端排队延时所采用的拓扑



(a) 1Gbps 网络环境下　　(b) 10Gbps 网络环境下

图 1.5　终端排队延时

带宽，所以我们关闭了 TSO（TCP Segment Offload）和 GRO（Generic Receive Offload）。在后文中如未做特别说明，我们在 1Gbps 网络中均关闭了 TSO 和 GRO。数据包的大小最大为 MTU。从图中可以看出，在没有背景流干扰时，Memcached 的请求延时大约为 0.2ms。而当有一条大流干扰时，Memcached 在终端的排队延时大约为 2.8ms；两条时大约为 5.0ms; 四条时大约为 11.2ms。值得注意的是，当有四条大流干扰 Memcached 时，有一部分请求的终端排队延时达到了 12ms，这是在该实验环境下可以达到的终端排队延时的最大值（$MTU \times 1000/1Gbps \approx 12ms$）。

在 10Gbps 网络环境下，由于必须要使用 offload 特性才能充分利用带宽 [27,28]，所以我们在 PM_1 与 PM_2 上同时开启了 TSO 和 GRO。在后文中如未做特别说明，我们在 10Gbps 网络中均开启了 TSO 和 GRO。此时，在 QDisc Queue 排队的数据包的大小最大为 64KB，所以导致了更加严重的终端排队延时。如图 1.5(b) 所示，当有四条大流干扰 Memcached 时，终端排队延时的 90% 分位值大约为 9.0ms；而有八条大流时，达到了 16ms。由于虚拟化技术与容器技术的发展，同一台物理机上可以实例化几十台虚拟机或者容器，大流的并发度将会更大，如果没有合适的控制算法，在终端的排队延时甚至会达到 50ms（$64KB \times 1000/10Gbps \approx 50ms$）。此外，由于终端排队延时与交换机排队延时是相互独立的两部分，数据包在终端经历了如此严重的排队延时后，还有可能再次在交换机中经历排队，这将会严重影响应用的性能和用户体验。因此，终端排队延时是一个亟待解决的问题。

1.2 现有方案存在的问题

现有的流调度算法要解决的问题主要是 1）延时敏感型应用的延时保证；2）优化吞吐密集型应用的平均流完成时间；3）交换机排队延时三个问题中的一个或两个，而且大部分调度算法都需要假设数据流的相关信息已知，如数据流大小和请求时间限制等。由于多数算法都是在数据包进入队列排队后，通过对数据包进行丢包、打标或是多优先级队列的方式进行调度的方法来提供解决方案，所以我们根据队列数量将已有的算法分成了两类：单队列算法和多队列算法。

1.2.1 单队列算法

pfifo[29] 是一种最基本的单队列调度算法。当排队过长，队列无法容纳更多的数据包时，再到达队列的数据包将会被丢弃。这种方案会在网络拥塞严重时，导致非常大的排队延时。RED[30] 和 CoDel[31] 是两种比较常用的 AQMs，它们在队列长度或是排队延时超过一个阈值后，直接丢弃数据包。尽管它们能有效降低队列长度，减小排队延时，但如果被丢弃的数据包来自于延时敏感型应用，则反而会对这类应用的性能产生严重影响。DCTCP[5] 通过 ECN 打标的方式判断网络中的拥塞情况，从而对拥塞窗口大小做出相应的修改，降低发送端的发送速率，从而减小队列的排队延时。但当数据流的并发度较大时，DCTCP 无法有效降低队列长度 [32,33]。由于所有数据流共享一个队列，单队列调度算法无法优化平均流完成时间。

1.2.2 多队列算法

为了保证每条数据流的调度公平性，fq_CoDel[34] 使用哈希算法根据五元组信息将数据流分配到不同的队列中（可能发生哈希冲突），每个队列都使用 CoDel 算法控制数据包的排队延时。由于需要大量队列的支持，fq_CoDel 很少在商用交换机中使用。

现在的商用交换机通常支持多优先级的多队列模式，但应用出于自身的利益考虑，通常都会选择最高优先级，导致多优先级队列退化成单队列。为了能利用多优先级队列的优势，QJUMP[9] 将应用选择的优先级与其能获得的带宽耦合在一起：选择高优先级的应用只能获得较低的带宽，而选择低优先级的应用可以获得较高的带宽。应用通过权衡自身的需求选择最合适的优先级。QJUMP 将优先级的选择权完全交给应用，可能会导致应用无所适从，这也导致了 QJUMP 无法根据数据流大小分配优先级来优化平均流完成时间。

PDQ[6] 和 pFabric[11] 虽然能近似实现平均流完成时间最小化，但它们都需要交换机的支持，并且需要假设数据流大小已知。PIAS[12] 不需要假设数据流大小已知而能近似实现 SJF（Shortest Job First）。PIAS 根据数据流已经发送的数据量分配优先级。当数据流刚建立时，PIAS 为其分配最高优先级；随着数据流发送的数据量越来越大，PIAS 逐渐降低其优先级。虽然 PIAS 能近似实现平均流完成时间最小化，但它会降低那些使

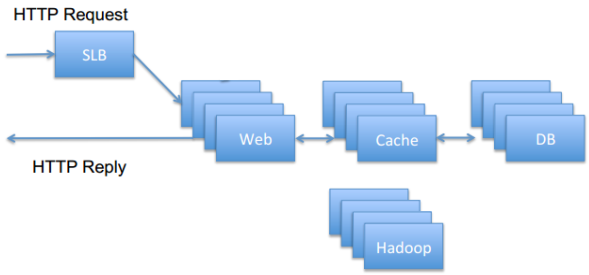


图 1.6　Facebook 数据中心服务类型

用 TCP 长连接进行数据传输的延时敏感型应用的性能（§2.2.2）。

Karuna[10] 可以在保证延时敏感型应用的低延时的同时优化平均流完成时间，但它需要假设延时敏感型应用的请求时间限制已知。pFabric[11] 与 D^3[8] 也同样需要假设延时敏感型应用的请求时间限制已知。

1.3　数据中心的流量特征

数据中心的流量特征决定了数据中心的拓扑结构、拥塞控制算法和服务质量保证算法等的设计。通常来说，不同数据中心由于运行的应用的不同，其流量特征也不尽相同，比如 Microsoft 数据中心的流量 [1,2] 与 Facebook 数据中心 [4] 的流量就有较大差异。本论文主要分析 Facebook 数据中心的流量特征 [4]，并说明在其他数据中心，如 Morgan Stanley[3]、Aliyun[35] 和 Amazon Web Services[36] 中也有类似的流量特征。

1.3.1　数据包大小分布

Facebook 数据中心主要运行着两类应用，一种服务于在线社交网络，一种用于离线分析数据 [4]，如图 1.6 所示。当一个社交网络的 HTTP 请求到达数据中心后，请求最先到达软件负载均衡器（software load balancer，SLB），之后这个请求会转发给一台网页服务器（web servers）。网页服务器直接从缓存服务器（cache servers）中读取数据，当发生 cache miss 时，缓存服务器会从数据库中请求数据。离线分析数据使用的应用是 Hadoop[37]。Hadoop 并不直接参与服务用户的请求，而是对数据做离线分析。因此，我们将 Facebook 数据中心的服务器分成四类：web servers，cache leaders，cache followers，Hadoop。

图 1.7 展示的是 Facebook 数据中心内数据包大小的分布 [4]。从图中可以看出，Hadoop 流量呈两极分布，基本上所有的数据包大小都是 MTU（1500 bytes）或是 TCP ACKs。其他服务所产生的数据包的大小分布范围较广，而且大部分都是小包：大约 85% 的数据包小于 400 Bytes，而充分使用 MTU 的数据包只占 5% 左右。

我们在 1Gbps 和 10Gbps 网络环境中重做了该实验，数据包大小在数据包进入发送端的物理网卡前抓取。我们使用 Memcached 模拟 web servers，cache leaders，cache followers 三类服务产生的流量。此外，我们根据 Hadoop 流大小将数据流分成三类：小

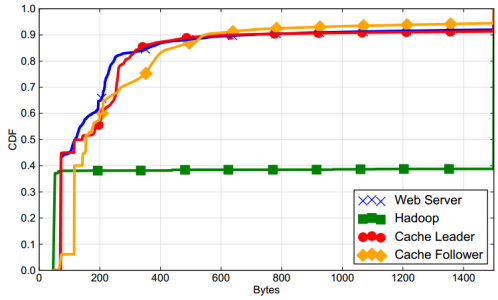
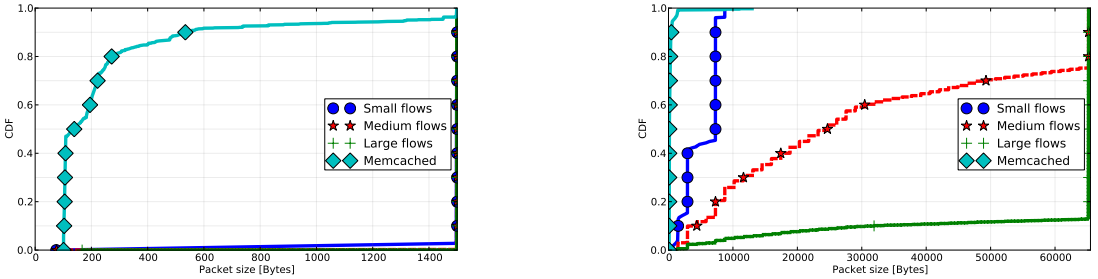


图 1.7 Facebook 数据中心中数据包大小分布



(a) 1Gbps 网络环境下 (b) 10Gbps 网络环境下

图 1.8 数据包大小分布

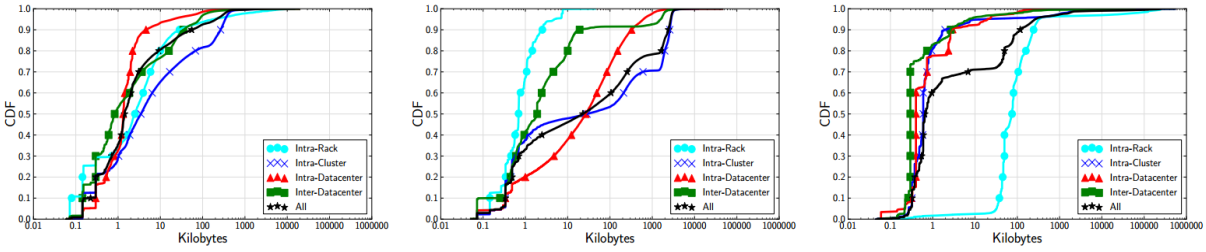
流 $(0, 100KB]$, 中流 $(100KB, 10MB]$ 和大流 $(10MB, \infty)$。数据包大小的统计中不包含 TCP ACKs。

1Gbps 网络的数据包大小分布如图 1.8(a) 所示，三种数据流的数据包大小绝大部分都是 MTU，而 Memcached 的数据包中 ~95% 都小于 MTU。而在 10Gbps 网络中，由于 offload 特性的影响，四种数据流的数据包大小分布区别更加明显，如图 1.8(b) 所示。Memcached 的数据包中仍有超过 95% 的数据包小于 MTU；而小流的数据包大小主要集中在 $[MTU, 10KB]$; 中流的数据包大小在 $[MTU, 64KB)$ 内分布比较均匀，约有 25% 的数据包大小为 64KB；而大流中约有 90% 的数据包都是 64KB。

因此，我们认为，在 Facebook 数据中心内，延时敏感型应用产生的流量大部分都是小包，不论是否启用 offload 特性，小于 MTU 的数据包都占据绝大部分。而吞吐密集型应用产生的流量趋向于大包，在没有 Offload 特性时，绝大部分数据包大小都是 MTU；而当有 offload 特性时，绝大部分数据包大小都大于 MTU，且数据流大小越大，其产生的数据包越大。

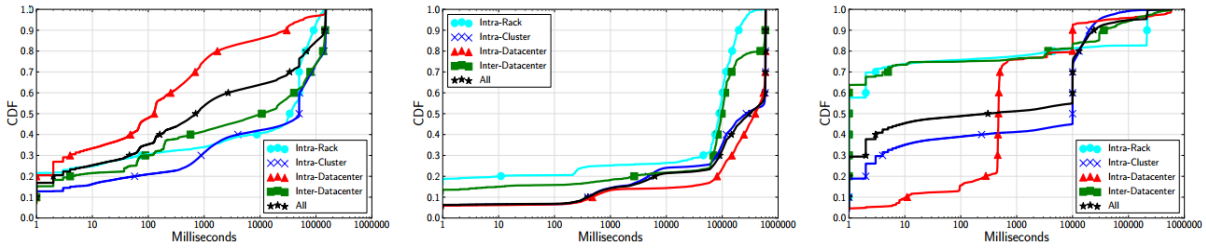
1.3.2 数据流大小和持续时间分布

Facebook 数据中心采用传统的多层网络，多台物理机构成一个机架（Rack），多个机架构成一个集群（Cluster），多个集群构成一个数据中心。因此，我们可以将数据流分成四类：Intra-Rack，Intra-Cluster，Intra-Datacenter，Inter-Datacenter。这四种数据流的流大小分布如图 1.9 所示 [4]。对于 web servers 产生的流量，其数据流大小的 90% 分位值



(a) Web servers (b) Cache follower (c) Hadoop

图 1.9 Facebook 数据中心中数据流大小分布



(a) Web servers (b) Cache follower (c) Hadoop

图 1.10 Facebook 数据中心中数据流持续时间分布

大约为 100KB；cache followers 产生的流量中，90% 分位值约为 2000KB；而对于 Hadoop 这种吞吐密集型应用，其数据流大小的 90% 分位值约为 100KB。所以，如果从数据流大小分布的角度比较延时敏感型应用和吞吐密集型应用，我们并没有发现明显的区别。

虽然在延时敏感型应用产生的流量中，小包占绝大部分，但其数据流大小却并不比吞吐密集型应用产生的数据流要小，甚至还要大，其原因是 Facebook 数据中心内的延时敏感型应用采用 TCP 长连接的方式降低延时。虽然这些长连接的吞吐量较低，但持续时间长，导致其数据流大小仍然较大。四种数据流的流持续时间如图 1.10 所示 [4]。cache followers 主要采用连接池（connection pooling）的方案，即 TCP 长连接。从图 1.10(b) 可以看出，只有 30% 的数据流的持续时间不足 100s，而超过 40% 的流大于 10 分钟。由于抓取时间只有 10 分钟，所以我们推测，有一部分长连接的持续时间会更长，也就是说，其数据流大小会更大。

1.3.3 数据中心数据流分类

运行在数据中心内的应用通常被分成两类：延时敏感型应用和吞吐密集型应用。从 Facebook 数据中心内的流量特征可以看出，这两类应用产生的数据流在流大小上没有明显区分，但在数据包大小的分布上差异较大。因此，我们将数据流也分成两类：低吞吐数据流（flows with low throughput, FLT）和高吞吐数据流（flows with high throughput, FHT）。这两种流的直接区别在于其数据包大小的分布上，而与其数据流的持续时间和流大小均不直接相关。从 Facebook 数据中心的流量特征中，我们发现，延时敏感型应用更倾向于产生 FLT，而吞吐密集型应用产生的数据流大多是 FHT，我们可以利用这个

特征区分应用类型，为应用提供服务质量保证。也就是说，我们要在尽可能保证 FLT 低延时的同时优化 FHT 的平均流完成时间。

FLT 和 FHT 这两种数据流在很多数据中心内共存。如在 Aliyun[35] 和 Amazon Web Services[36] 中存在大量物联网应用。很多物联网应用为了提升用户体验，通常采用 TCP 长连接传输数据。比如在家电物联网应用中，用户通过手机远程控制家用电器，而家用电器也需要主动向用户推送自身的状态信息，所以通常采用长连接设计。由于需要与用户进行直接交互，这种应用就是一种典型的延时敏感型应用。一般情况下，这种应用拥有专用的协议，所以用户与家用电器交互时所发送的数据包不会太大。因此，这类应用产生的数据流就是我们所定义的 FLT。微信 [38] 的文字传输使用的数据流也属于 FLT。FLT 倾向于要求提供低延时保证，但在实际中经常会受到 FHT 的干扰，如视频、文件传输等。由于在一些情况下，数据中心服务提供商无法在传输开始前获取到应用的相关信息，如数据流大小等，因此，我们的目标是 1）不假设相关数据流信息已知；2）尽可能降低 FLT 的延时；3）优化 FHT 的平均流完成时间。

1.4 本论文的工作

1.4.1 Panda 调度算法

本论文提出的 Panda 调度算法能在不需要预知数据流信息的前提下，同时保证 FLT 的低延时与优化 FHT 的平均流完成时间。Panda 通过分析数据中心的流量特征，找到了一个区分阈值，将所有数据包分成两组：大包和小包。该阈值能保证 FLT 产生的流量中绝大部分都属于小包，而 FHT 产生的流量中大部分都属于大包。Panda 以数据流为粒度进行调度。为了记录数据流的状态，Panda 为每条 TCP 流分配了一个计数器。在数据包离开终端前，Panda 根据数据包大小更新其对应的数据流的计数器，并根据计数器的值为数据流分配优先级。通过配合多优先级队列，Panda 可以实现上述目标。

Panda 在实际应用中会遇到三个挑战：1）如何选取区分数据包的阈值；2）如何根据数据包大小更新数据流的计数器；3）如何根据计数器的值为数据流分配优先级。

我们通过建立一个数学模型来解决第一个挑战。当 FLT 产生的数据包中小包大小过小时，会导致小包所占的流量比例被只占少数的大包掩盖。为了解决这个问题，我们将数据包大小映射成数据包在离开物理网卡后会被切分成的数据包的数量。假如数据包的大小分别为 100、1400、2000 和 10000，则使用映射函数后的值分别为 1、1、2 和 7。然后，我们得到 FLT 与 FHT 两种类型的应用的小包数量所占比例，通过最大化 FLT 与 FHT 的小包数量所占比例之差，我们可以得到最优的大包小包划分阈值。

对于第二个挑战，我们选择了一个补偿函数来根据数据包大小更新计数器。更新函数的选择会直接影响 Panda 的性能，如果选择不恰当，会大大缩短 FLT 能停留在高优先级的时间。当数据包为大包时，说明该数据流更有可能为 FHT，所以 Panda 增加其计数器的值；而当数据包为小包时，该数据流倾向于属于 FLT，所以 Panda 降低其计数器的

值，对其进行补偿。通过这种更新计数器的方式，Panda 可以大幅度延长 FLT 被分配高优先级的时间；此外，还可以保证数据流的优先级只与数据包大小分布有关，而与数据流大小无关。

我们建立一个最优化模型来解决第三个挑战。TCP 数据流刚建立时，其计数器的值为 0，所以 Panda 为其分配最高优先级。随着计数器值的增加，超过某个阈值后，Panda 降低其对应的数据流的优先级；再超过另外一个阈值后，Panda 继续降低其优先级；直到最低优先级。我们根据 FHT 的历史数据流大小分布可以推导出近似最优的优先级变化阈值。

我们在 Linux 平台上实现了一个 Panda 的原型系统。Panda 作为一个内核模块被安装在 TCP/IP 协议栈与 Linux TC 之间，其对于应用来说完全透明。为了方便交换机和 AQMs 获取数据流的优先级，保证 FLT 的低延时与优化 FHT 的平均流完成时间，Panda 将优先级信息填入数据包的 DSCP 字段。此外，为了进一步降低 FHT 的平均流完成时间，我们推荐 Panda 使用 DCTCP 作为传输层协议。

我们分别在 1Gbps 和 10Gbps 网络环境下评测了 Panda 的性能，使用的延时敏感型应用是 Memcached，吞吐密集型应用是一种真实的网页搜索流量负载 [5]。实验数据表明，Panda 能有效降低 FLT 的响应延时。当 FLT 数据流大小较大时，与 PIAS 比较，在 1Gbps 网络环境下，Panda 能降低请求延时的 99% 分位值达 ~50%（使用 DCTCP）和 ~97%（使用 CUBIC）；在 10Gbps 网络环境下，Panda 能降低请求延时的 99% 分位值达 ~87%（使用 CUBIC）。此外，在优化 FHT 平均流完成时间方面，Panda 与 PIAS 具有基本一致的性能。

1.4.2 本论文贡献

现有的数据流调度方案中，多数都会假设数据流的某些信息已知，如应用类型、数据流大小等。此外，大部分的方案只解决了两个问题中的一个，或是保证 FLT 的低延时，或是优化 FHT 的平均流完成时间。本论文提出的方案（Panda）可以同时解决这两个问题。通过分析数据中心内的流量特征，Panda 也不需要预知任何有关数据流的信息。本论文的贡献包括：

(1) 本论文通过分析 Facebook 数据中心内的流量特征，找出了延时敏感型应用和吞吐密集型应用所产生的数据包在进入终端物理网卡前，在有无 offload 特性下所对应的数据包大小分布。
   - 延时敏感型应用产生的数据包大部分是小包，而吞吐密集型应用产生的数据包大部分都不小于 MTU；
   - 延时敏感型应用与吞吐密集型应用所产生的数据流大小并无明显区别。

(2) 本论文提出了一种有效的流调度算法 Panda，在不需要提前预知数据流信息的前提下，能为 FLT 和 FHT 两种数据流同时提供服务质量保证。

(3) 我们在 Linux 平台上实现了一个 Panda 原型系统，并在终端实现一个多优先级队列 AQM。实验数据表明，Panda 可以保证 FLT 的低延时，并能优化 FHT 的平均流完成时间。

1.5 论文的组织

本论文共有六章内容，讨论了如何在数据中心内，不需要预知数据流信息的前提下，同时保证延时敏感型应用和吞吐密集型应用的服务质量。

第二章介绍了现有的几种有代表性的服务质量保证方案，包括应用于终端的 AQMs，如 RED[30] 和 CoDel[31]，在数据中心内广泛应用的 DCTCP[5]，还有 QJUMP[9] 和 PIAS[12]；并指出了这些方案在解决我们提出的目标时所存在的不足。

第三章详细描述了 Panda 的设计。我们首先介绍了 Panda 的三个组成模块：数据收集，离线建模和在线分类，然后分别详细说明了每个模块的细节。

第四章讨论了 Panda 在 Linux 环境下的实现方案，并介绍了 Panda 如何与多优先级队列配合来同时实现两种应用的服务质量保证。

第五章使用 FLT 的延时和 FHT 的平均流完成时间两个指标对 Panda 在 1Gbps 网络环境和 10Gbps 网络环境下的性能进行了评测。

第六章总结了本论文的工作，并提出了下一步的研究计划。

第二章　相关工作

很多调度算法的设计都是为了实现数据中心内高吞吐低延时的目标。这些调度算法的目标大致可以分为三类：为了降低数据包的延时，为了实现高吞吐，为了降低数据流完成时间。由于本论文主要解决的问题是如何保证延时敏感型应用的低延时和优化吞吐密集型应用的平均流完成时间，所以本论文对相关工作的讨论主要集中在降低数据包的延时和降低流完成时间这两类调度算法。

2.1　降低延时的调度算法

2.1.1　随机早期检测 RED

在要求高吞吐低延时的高速网络中，为了避免流量的突发带来的瞬时拥塞而导致丢包的问题，传统的方法是在网关（gateway）上设计一个较大的缓冲区来容纳突发流量。这种方案虽然表面上能减少丢包率，但会带来两个新的问题：1）较大的缓冲区增大了数据包在网关所经历的平均排队延时；2）很多网络协议都是通过检测丢包来判断网络中是否出现了拥塞，从而降低发送速率，但因为网关的缓冲区缓存了大量数据包，发送端不能及时发现网络中的拥塞，减小拥塞窗口，所以突发的流量会越来越严重，反而导致了更严重的排队延时和丢包率。

RED[30]（Random Early Detection）通过采用在网关处随机丢包的方式降低发送端的发送速率的方式来降低网络的拥塞程度，减小排队延时。RED 在网关进行拥塞检测的理由有三点：1）网关能可靠区分传播延迟和排队延迟，因为排队延迟在网关产生，而且对协议栈透明；2）网关对排队延时行为有一个较全面的认识，能分析导致排队产生的原因和控制导致排队的因素，如队列长度和链路带宽等；3）网关由多条数据流共享，在网关处可以同时控制多条流的丢包率，调整它们的发送速率，而不需要对发送端或者接收端进行任何修改。

RED 需要设置两个队列长度参数 min_{th}，max_{th} 和一个丢包概率参数 max_p。RED 的丢包算法如算法 1 所示。当一个数据包到达队列时，先计算队列的平均队列长度 avg。如果 $avg < min_{th}$，说明此时网络并没有拥塞，不需要降低拥塞窗口的大小，所以不需要对数据包做任何操作。当 $min_{th} \le avg < max_{th}$ 时，先计算丢包概率 p_a，然后根据 p_a 丢弃数据包。p_a 与 avg 正相关，即 avg 越大，p_a 越大。而当 $max_{th} \le avg$ 时，说明此时网络的拥塞情况非常严重，需要通过迅速丢包的方式降低发送端的发送速率，减小排队延时。所以，此时直接丢包，即 $p_a = 1$。

RED 没有使用固定概率丢包，而是根据平均队列长度与两个队列长度阈值的函数确定丢包概率，其原因是平均队列长度本身就在一定程度上反映了网络的拥塞状态。当

Algorithm 1 Random Early Detection 算法

```
for each packet arrival do
    calculate the average queue size avg
    if min_th ≤ avg < max_th then
        calculate probability p_a
        mark the packet with p_a
    else if max_th ≤ avg then
        mark the packet
    end if
end for
```

平均队列过长时，说明网络拥塞严重，所以丢包的概率需要更大一些；而当平均队列较小时，说明网络拥塞情况不太严重，只需要少量比例的丢包即可。丢包概率的计算如公式 2-1所示:

$$p_a \leftarrow max_p(avg - min_{th})/(max_{th} - min_{th}) \tag{2-1}$$

当数据包到达队列后，RED 使用 EWMA[39]（Exponential Weighted Moving Average）计算平均队列长度。假设权值衰减系数为 α，q_n 和 avg_n 分别表示第 n 个数据包到达时的实际队列长度和平均队列长度。根据 EWMA，avg_n 的迭代表达式如公式 2-2所示:

$$avg_n = \begin{cases} q_1 & ,\text{n=1} \\ \alpha q_n + (1-\alpha) avg_{n-1} & ,\text{n>1} \end{cases} \tag{2-2}$$

RED 的优点在于能有效降低排队延时和解决流量突发的问题。但其缺点也很明显，RED 需要设置两个队列长度阈值 min_{th} 和 max_{th}。由于没有一个精确计算的方法，导致 RED 在实际部署中很难把控参数的设置，如果参数设置不合理，反而会导致网络性能降低。

2.1.2 CoDel

尽管 RED 算法简单，且能有效降低队列长度，但它的配置参数与实际系统密切相关。RED 根据队列长度判断数据包是否需要丢弃，由于在参数设置上没有合适的参考，在一些情形下，RED 的性能并不理想 [31]。此外，[40] 认为队列长度并不是判断网络中是否出现了拥塞的一个很好的指标。Kathleen[31] 从另一个角度分析了队列产生的原因，并将排队分成了两类：好的与坏的。为了吸收猝发流量而产生的队列被认为是好的队列，而另一种被称为 standing queue 的队列，则被认为是坏的队列。这种队列由于拥塞窗口大小与链路带宽的不匹配而导致，而在当调整拥塞窗口大小，使之与链路带宽匹配之后，这种队列依然会存在，且很难被消除。因此，它们只会带来不必要的排队延时，而对吞

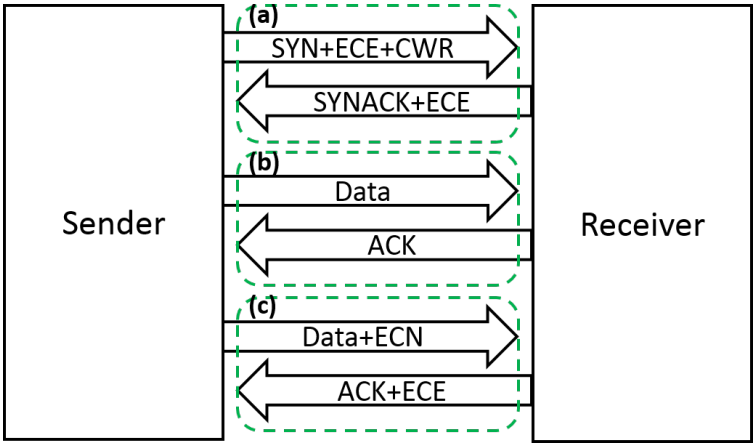


图 2.1 ECN 通知机制

吐没有任何好处。Kathleen 认为 standing queue 是造成排队延时的元凶。因此，CoDel 的目的就是为了保证好的队列不受影响，并尽可能降低坏的队列的队列长度。

为了保证参数与系统无关，CoDel 没有使用列队长度作为判断网络拥塞的依据，而是使用每个数据包的实际排队时间（packet-sojourn time）。用户真正关心的直接性能指标是延时，而当使用队列长度作为控制排队延时的指标时，则需要考虑网络的链路带宽等实际因素。也就是说，即使用户有明确的排队延时限制，也需要根据实际网络带宽修改队列长度阈值。此外，使用列队长度作为判断网络拥塞的方案无法分辨好的队列与坏的队列。虽然该方案能较好地控制坏的队列的长度，但同时也会限制好的队列的长度，影响性能。

CoDel 在数据包进入队列（*enqueue*）时，给数据包打上时间戳；当数据包离开队列（*dequeue*）时，得到数据包在该队列的排队时间。如果排队时间超过阈值（*target*，默认值为 5ms），且持续时间超过持续时间阈值（*interval*，默认值为 100ms）时，则需要考虑是否丢弃该数据包。CoDel 设置持续时间阈值的目的是为了区分好的队列与坏的队列。CoDel 的伪代码见附录 A[41]。

2.1.3 ECN 拥塞通知机制

RED 和 CoDel 通过丢包的方式降低数据包的排队延时；与此同时，丢包可以作为网络拥塞的信号，使得拥塞窗口减小，从而降低数据流的发送速率，进一步降低队列长度。但丢包会直接导致重传，延时敏感型应用无法接受以重传为代价降低排队延时。因此，数据中心内通常会使用 ECN[42]（Explicit Congestion Notification）机制作为网络拥塞的信号，而不是直接丢包。

ECN 机制通过对数据包打标而不是丢包的方式来实现对网络拥塞的通知。当网络中出现拥塞时，数据包会在瓶颈链路处形成队列。而当队列长度超过一个阈值后，为了减少排队延时，降低发送端的发送速率，交换机会在数据包上打标。

ECN 拥塞通知机制如图 2.1 所示。如图 2.1 的（a）过程所示，当 TCP 连接建立时，

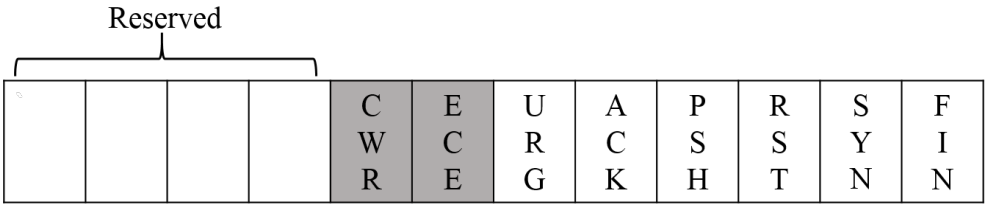


图 2.2　TCP 控制字段

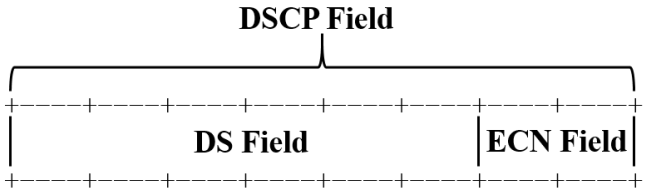


图 2.3　DSCP 字段

发送端会将其 SYN 包的 TCP 控制字段的 ECE 位和 CWR 位设置为 1（TCP 控制字段如图 2.2 所示）；接收端收到 SYN 包后，会将其返回的 SYNACK 的 TCP 控制字段的 ECE 置 1；发送端收到 SYNACK 后，确认该 TCP 连接支持 ECN 拥塞通知机制（ECN-Capable Transport），从而在之后发送的数据包中，将 IP 的 ECN 字段置为 01 或者 10。ECN 字段如图 2.3 所示。如果数据包在传输路径中没有遇到拥塞，如图 2.1 的（b）过程所示，数据包不会被标记拥塞，接收端收到数据包后返回 ACK，与原机制一致。而当数据包在路径中遇到了拥塞时（队列长度超过阈值），如图 2.1 的（c）过程所示，交换机会将数据包的 ECN 字段置成 11，表明该数据包在路径中遇到了拥塞（CE）。接收端收到该数据包后，会将 ACK 的 ECE 字段置 1，通知发送端遇到了拥塞。在标准 ECN 中，发送端会将其拥塞窗口大小减半。

需要注意的是，ECN 通知机制需要 TCP/IP 协议栈的支持。可以通过 *sysctl* 接口设置*/proc/sys/net/ipv4/tcp_ecn* 选项配置终端是否支持 ECN。该选项有三个可选值：0、1、2，0 表示不支持 ECN，1 表示支持 ECN，2 表示不确定。此外，ECN 也需要交换机的支持。

ECN 拥塞通知机制的优点是不需要丢包就可以判断网络中出现了拥塞，从而通知发送端降低发送速率。改进后的 RED 和 CoDel 也支持 ECN 通知机制，当队列长度超过阈值时，通过对数据包进行 ECN 打标，而不是直接丢包的方式通知网络拥塞。

2.1.4　DCTCP 拥塞控制算法

DCTCP[5] 是一种广泛应用于数据中心环境的 TCP 变种，它利用 ECN 拥塞通知机制，根据 ECN 的打标比例评估网络的拥塞程度，调整拥塞窗口大小。与传统的 TCP 比较，DCTCP 有四个显著优点:1) 对于吞吐密集的数据流，DCTCP 在保持同样或者更高的吞吐时却可以少用 90% 的 buffer 空间 [5]。由于数据中心普遍使用浅 buffer 的廉价交换机，所以这一优点使得 DCTCP 能够很好地适用于数据中心网络环境；2）由于使用很

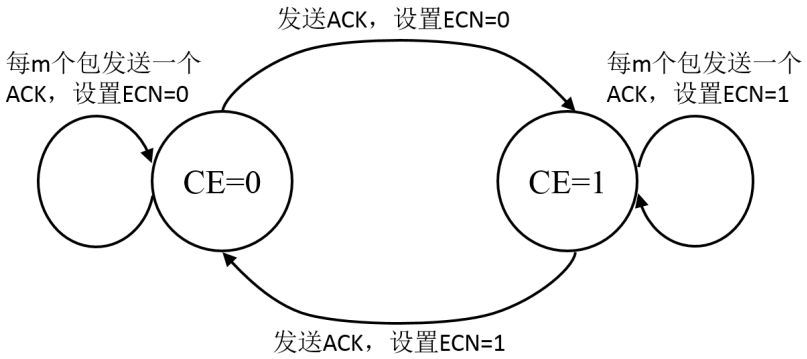


图 2.4 DCTCP 状态机

少的 buffer 空间，使得 DCTCP 可以容忍更大的流量突发；3），由于 DCTCP 可以有效降低交换机的队列长度，因此保证了延时敏感型数据流的低延迟；4）DCTCP 具有更好的收敛性。

DCTCP 保留了标准 TCP 的慢启动算法与拥塞避免算法，故当网络中没有拥塞时，DCTCP 的窗口增长方式与 TCP 一致。而当网络中发生拥塞时，DCTCP 对拥塞窗口的调整与标准 TCP 的拥塞算法不同，它根据 ECN 的打标比例估计网络的拥塞程度，从而调整拥塞窗口大小。如果 ECN 打标比例很高，说明网络拥塞严重，因此需要较大幅度降低拥塞窗口的大小；而当 ECN 打标比例较低时，说明网络只是略有拥塞，小幅度降低拥塞窗口即可。DCTCP 的拥塞窗口更新算法如公式 2-1 所示：

$$cwnd \leftarrow cwnd \times (1 - \alpha/2) \tag{2-3}$$

α 的初始值为 1，每次更新拥塞窗口大小前，需要对其值进行更新。α 的更新公式如公式 2-2 所示：

$$\alpha \leftarrow (1 - g) \times \alpha + g \times F \tag{2-4}$$

其中，F 为一个发送窗口中打了 CE 标记的数据包占该窗口内发送的数据包的比例。g 为采样因子，用于对 α 的值做平滑处理，其取值区间为 $(0, 1)$。根据 [5] 的推荐，g 通常取值为 1/16。每次成功发送一个窗口的数据后，DCTCP 都会根据这个窗口的拥塞状况来更新网络的总体拥塞状况，即 α 的值，然后调整拥塞窗口的大小。

DCTCP 的拥塞反馈机制可以用一个有两个状态的状态机表示，如图 2.4 所示。DCTCP 采用延迟 ACK（Delayed ACK），当所收到数据包的 ECN 字段一直为 CE 状态时，接收端每收到 m（[5] 推荐 $m = 2$）个数据包回复一个 ACK（带 ECE 标记）；当状态发生变化时，立即回复一个 ACK，以此来减少 ACK 为数据中心网络带来的负载。

2.2 降低平均流完成时间的调度算法

RED，CoDel 和 DCTCP 都是通过限制队列长度来降低数据包的排队延时。这些算法主要针对的对象是单队列，它们对延时敏感型应用产生的流量和吞吐密集型应用产生的流量不作区分，即这些流量共享同一个队列。图 1.5 说明，当吞吐密集型应用与延时敏感型应用共享队列时，吞吐密集型应用产生的流量会在队列产生排队，从而导致延时敏感型应用的流量遭受较大的排队延时。

如果系统中存在多个队列，而且这两种应用不共享同一队列，这样就能保证吞吐密集型应用不会干扰延时敏感型应用的性能，这种方案被称为 MLFQ（Multiple Level Feedback Queue）。通过给多队列分配不同的优先级，进入高优先级队列的流量将会被优先服务，这样就可以保证具有高优先级的流量的低延时。由于短流优先策略（Shortest Remaining Processing Time，SRPT）可以最优化平均流完成时间 [6,11]，因此，如果我们能在传输开始前预知数据流的大小，我们便可以根据流大小分配优先级，最小化平均流完成时间。但在真实系统中，这些信息很有可能无法提前得到。此外，为了实现 SRPT，还需要一个中心调度器，但从中心调度器将优先级的信息传递给交换机的延时也将会无法避免 [43]。

2.2.1 QJUMP

如果系统能为延时敏感型应用产生的流量分配高优先级，而为吞吐密集型应用产生的流量分配低优先级，就能保证延时敏感型应用的低延时要求。此外，由于不需要为延时敏感型应用预留带宽，吞吐密集型应用便可以充分利用剩余的带宽。但这种方案的难题在于：如何确认哪有应用应该被分配高优先级。

数据中心内使用的交换机支持 IEEE 802.1Q 标准，它提供了 8 个优先级。但这 8 个优先级在实际中很少被使用，原因是从博弈论的角度分析，应用只有认为自己的服务最重要，才会得到更好的网络服务，尽管这是一个纳什均衡点 [44]，但并不是全局最优点。如果优先级由应用自己决定，那么 8 个优先级中只有最高优先级才有意义。比如，Memcached 的开发者当然会认为 Memcached 产生的流量在整个系统中最重要，那么它的优先级将会是 1（最高优先级）；而 Hadoop[37] 的开发者也会认为最高优先级应该分配给 Hadoop，尽管它产生的流量吞吐量较大，但它执行的任务更重要。假设网络中存在一个比较正直的应用，它的开发者觉得它产生的流量吞吐量很大，如果优先级太高的话，肯定会干扰其他延时敏感型应用，所以不应该被分配最高优先级。但结果将会是这个最正直的应用受到了系统最差的服务。由于正确的优先级分配会受到了惩罚，所以最终系统中一定会是所有的应用都选择最高优先级。

Matthew 等人 [9] 认为，多优先级队列无法被系统实际使用的原因主要是因为系统对选择高优先级的应用没有任何限制，导致所有应用都会选择最高优先级。QJUMP[9] 决定耦合应用的优先级与吞吐量。由于优先级与延时相关，QJUMP 的核心思想是，优

先级仍由应用决定，但应用选择的优先级越高，其能分配到的带宽就越低。所以，对于延时敏感型应用，由于其要求低延时，且对带宽的要求不高，所以这类应用会选择高优先级；而对于吞吐密集型应用，如果它们仍然选择高优先级，系统便会对它们的带宽进行限制，迫使它们选择低优先级。通过这种惩罚机制，QJUMP 便可以利用多优先级队列的优点保证不用应用类型的服务质量。

QJUMP 根据系统参数，如最大数据包大小 P，瓶颈链路带宽 R 和网络中所有终端数量 n，得到了系统在没有任何排队延时下的最大端到端延时 D：

$$D \leq n \times \frac{P}{R} + \epsilon \tag{2-5}$$

其中 ϵ 表示网络中由交换机导致的累积处理延时。如果能保证每台终端在一个周期（epoch）内，即每 D 时间发送的数据包数量不超过一个，那么整个系统中的端到端延时将会不大于 D。为了保证所有终端都在同一个周期发送数据包，系统不得不使用全局时钟同步（synchronization）机制。尽管 PTP 同步 [19] 可以实现系统的全局时钟同步，但这种方案并未得到广泛支持。因此，QJUMP 使用另一种方案实现了系统的均步（mesochronous）。QJUMP 将发包周期增大一倍，即使不需要同步，也可保证当每台终端在一个周期内发送的数据包不超过一个时，系统的端到端延时不超过 D。因此，QJUMP 将每台终端的周期设置为：

$$epoch = 2n \times \frac{P}{R} + \epsilon \tag{2-6}$$

使用分布式的方式保证端到端延时是 QJUMP 的一个重要优点。

但这种方案会严重影响网络的实际带宽。每台终端可以利用的带宽如公式 2-7所示。

$$throughput = \frac{P}{epoch} \approx \frac{R}{2n} \tag{2-7}$$

假设网络中共有 1000 台终端，链路带宽为 10Gbps，但每台终端可以使用的带宽却不足 5Mbps。导致链路带宽无法被使用的原因主要是因为假设太过于保守。公式 2-5假设所有的终端同时向一台终端发送数据，但这在真实环境中属于小概率事件。此外，虽然所有终端在一个周期内发送一个数据包可以保证所有数据包的低延时，但有些应用却可以容忍一定程度的排队延时。因此，QJUMP 引入了一个吞吐因子（throughput factor）f，将系统中的实际终端数量 n 转换成了相对终端数量 n':

$$n' = \frac{n}{f}, where\ 1 \leq f \leq n \tag{2-8}$$

如果终端 i 选定的吞吐因子为 f_i，那么它能得到的带宽为

$$throughput \approx \frac{R}{2n'} = \frac{f_i R}{2n} \tag{2-9}$$

终端能获得的实际带宽与 f 成正比。

QJUMP 将吞吐因子与应用的优先级建立映射关系，且负相关。如果应用选择的优先级较高，其吞吐因子则较小，能获得的带宽较低；如果应用选择的优先级较低，则其吞吐因子较大，带宽也会较高。

QJUMP 通过惩罚机制将延时敏感型应用与吞吐密集型应用区分开，保证吞吐密集型应用产生的流量不干扰延时敏感型应用产生的流量。但 QJUMP 将选择的权利完全交给了应用，这会导致应用无所适从，不知该如何选择。由于交换机支持 8 个优先级，当应用属于两个极端时，即极端延时敏感型应用和极端吞吐密集型应用，应用的优先级选择很明显，选择 1 或 8 即可。但当应用属于中间类型时，由于没有具体的指导方案，有些应用甚至无法预知自己的预期吞吐量时，这很有可能会导致应用随机选择优先级。此外，由于优先级由应用决定，系统无法优化数据流的平均流完成时间。

2.2.2 PIAS

在数据中心的调度算法设计中，最重要的目标之一就是为了最小化数据流的平均流完成时间 [5,6,11,13,43,45]。虽然 SRPT 是最小化平均流完成时间的一种最优调度策略，但它需要预先知道数据流的大小信息，所以一些算法，比如 PDQ，pFabric 和 PASE[6,11,13]，通过假设数据流的大小已知来近似实现 SRPT。

但在一些情况下，数据流的大小信息很难提前获得，甚至无法提前获得。

HTTP 分块传输 HTTP 1.1[15] 开始支持数据分块传输，在整个数据生成过程中，动态产生的内容被分成多块发送。这种模式在数据中心中广泛使用。例如，应用可以使用分块传输将数据库的内容转存到 OpenStack Object Storage[46]。在分块传输中，一条数据流包含多块数据，数据流的总大小在传输开始时无法获得。

数据库请求响应 数据库系统中的请求响应是数据流大小无法预知的另一个例子。比如，在处理请求时，SQL servers[16] 直接发送它们收到的部分结果，而不是将这些结果缓存起来，等到处理请求结束后将所有的结果一起发送。数据流的大小信息在传输开始时同样无法获得。

流处理 在 Apache Storm 系统 [17] 中，在主节点给工作节点分配任务后，工作节点会先分析任务，然后与处理该任务相关的其他工作节点建立 TCP 长连接。数据在一个工作节点处理完成后，会直接发送到下一个处理节点继续处理。需要处理的数据的总量在流处理完成前一直未知。

实际限制 有些应用的数据流大小虽然可以获得，但缺乏相应的接口将这些信息传递到可供使用的地方。比如在 Hadoop 中，Mapper 会在 Reducer 开始获取数据之前将中间数据写入磁盘，因此，Reducer 可以提前获取到数据流的大小信息。但如何将该信息通过一个通用的 API 传递出来将会是另外一个难题。

PIAS[12] 的核心思想是不需要预知数据流大小而近似实现一个 SJF（Shortest Job

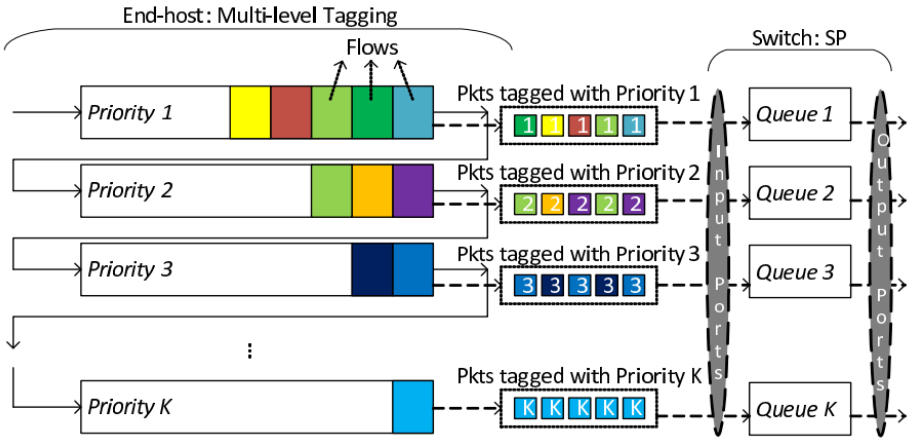


图 2.5 PIAS 的实现

First）算法。这类不需要提前知道数据流信息的调度算法被称为 non-clairvoyant scheduling[47]。PIAS 利用商用交换机中的多优先级队列实现了一个 MLFQ，如图 2.5 所示。当一条数据流新建立时，PIAS 为其分配最高优先级 1。PIAS 在终端给该数据流产生的数据包的 DSCP 字段打上优先级标记，当数据包离开终端进入交换机后，交换机通过识别数据包的 DSCP 字段将该数据包放入与其优先级对应的队列中。当数据包离开该交换机进入下一个交换机时，数据包仍能进入对应的队列中。由于是多优先级队列，具有高优先级的数据流产生的数据包将会被优先服务。随着该数据流产生的数据量逐渐增加，当超过一个阈值后，PIAS 将该数据流的优先级降低为 2；随着数据量继续增加，数据流的优先级将会逐渐降低，直到降为最低。

PIAS 的阈值设置与网络中的数据流大小分布有关，[12] 使用一个近似算法解出了每个优先级对应的阈值。因此，PIAS 会为小流（流大小较小的数据流）设置高优先级，而大流设置低优先级，从而近似实现了 SJF。此外，PIAS 使用 DCTCP 作为传输层协议降低队列长度，减小排队延时。

PIAS 虽然能在不需要预知数据流大小的情况下优化平均流完成时间，但这种方案会导致一些通过 TCP 长连接来保证低延时的延时敏感型应用遭受较大的延时。比如在 Facebook 数据中心中，Cache follower 产生的数据流有 40% 左右会超过 10 分钟，虽然其累计数据流大小较大，但仍然是延时敏感型应用。根据 PIAS 的策略，这类数据流会因为较大的数据流大小而被分配低优先级，从而导致它们产生的数据包经历很大的排队延时（§5）。

2.3 小结

本章主要讨论了几种相关的调度算法，包括三种单队列算法 RED、CoDel 与 DCTCP 和两种多优先级多队列算法 QJUMP 与 PIAS。尽管这些算法提出的背景都不太一样，它们却都能在一定程度上保证延时敏感型应用的低延时，或是优化吞吐密集型应用的平均流完成时间。但是，这些算法都无法在不需要预知数据流信息的前提下，同时满足这两

种应用的服务质量要求。如单队列算法无法优化平均流完成时间，而已有的多队列算法则无法保证延时敏感型应用的低延时要求。因此，数据中心仍然需要一种不需要假设数据流信息的调度算法来同时保证不同应用的服务质量要求。

第三章　Panda 的设计

数据中心调度算法设计的两个主要目标是：1）保证延时敏感型应用的低延时；2）最小化吞吐密集型应用的平均流完成时间。由于在数据中心内，一些应用的数据流信息，如流大小与请求时间限制等，在传输开始时很难或者根本无法获得，因此，我们的设计目标除了前两个外，还包括：3）不假设数据流大小信息与请求时间限制已知；4）易部署，即不需要对应用程序代码做任何修改，能直接运行在商用交换机上，与 TCP/IP 协议栈兼容，且调度算法对于应用程序完全透明。所以我们设计了 Panda 来同时实现这四个目标。

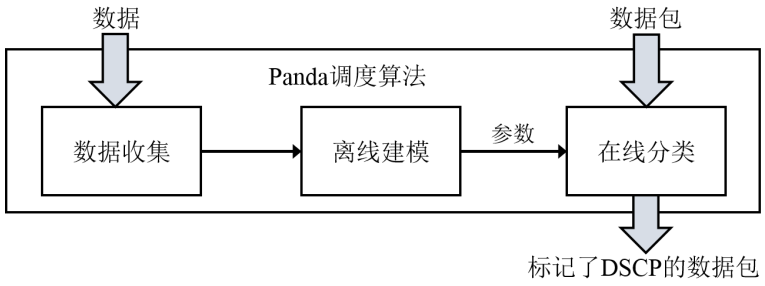


图 3.1　Panda 组成部分

如图 3.1 所示，Panda 包含三个组成部分：1）数据收集；2）离线建模；3）在线分类。数据收集模块通过收集历史数据，得到 FLT 与 FHT 的数据包大小分布。离线建模模块根据数据包大小的分布得到最优的参数，如数据包划分阈值和优先级划分阈值，并将这些参数传递给在线分类模块。在线分类模块划分大包与小包，并根据优先级划分阈值更新数据流的优先级，然后在数据包的 DSCP 字段上标记优先级。

3.1　数据收集

Panda 不需要预知数据流大小信息和请求时间限制，它通过 FLT 与 FHT 在数据包大小分布上的差异找到一个最能区分这两种类型的数据流的阈值。因此，为了同时保证这两种类型的数据流的服务质量，Panda 需要从数据中心内得到这两种数据流的数据包大小分布。

Panda 假设运行在数据中心内的应用是稳定的，即它们产生的流量在一段周期内，如一天或者一周，不会有太大的波动。因此，Panda 周期性的收集数据包大小信息，经过处理后，将这两种类型的数据流的数据包大小分布应用在下一个周期的离线模型中。离线模型根据最新的数据包大小分布更新参数，用于下一个周期的数据包划分与数据流的优先级划分。但具体收集和处理数据的方法不在本论文的讨论范围内。本论文假设 FLT

表 3.1　阈值划分模型中的参数说明

参数	含义
p	数据包
s_p	p 的数据包大小
T	划分阈值
f_i	数据流 i
$\|f_i\|$	数据流 i 的大小
N_i	f_i 的数据包数量
P_i	f_i 的数据包集合
P_i^s	f_i 中大小为 s 的数据包集合
$\|P_i\|$	f_i 的数据包数量
$\|P_i^s\|$	f_i 中大小为 s 的数据包数量
$Pr_i(s)$	f_i 中大小为 s 的数据包的比例
$g(\cdot)$	数据包大小映射函数
avg_i	f_i 中数据包的平均大小
$avg_i^s(T)$	f_i 中小包的平均大小
$avg_i^l(T)$	f_i 中大包的平均大小
$Pr_i^s(T)$	f_i 中小包的比例
$Pr_i^l(T)$	f_i 中大包的比例

和 FHT 的数据包大小分布函数已知。

3.2　离线建模

为了记录每条数据流的状态，来为数据流分配优先级，Panda 为每条数据流分配了一个计数器 counter。Panda 使用一个阈值 T 将数据包分成两类：大包和小包。当数据包为大包时，说明该数据流更有可能为 FHT，所以 Panda 增加其计数器的值；而当数据包为小包时，该数据流倾向于属于 FLT，所以 Panda 降低其计数器的值，对其进行补偿。由于 FLT 产生的流量中大部分都属于小包，因此 FLT 的计数器将会一直保持较小的值，也就是会被分配较高优先级。而对于 FHT，由于其产生的流量中大包占据绝大部分，所以 FHT 的计数器随着已经发送的数据量的增加越来越大，其优先级也越来越低。Panda 能保证 FLT 长时间处于高优先级，而 FHT 的优先级近似由其已经发送的数据量决定。

如何选择阈值 T 是 Panda 需要解决的第一个问题。如果阈值过小，会导致小包在 FLT 产生的流量中所占比例不高，Panda 无法保护 FLT 能长时间处于高优先级；如果阈值过大，会导致 Panda 无法区分 FHT 和 FLT，FHT 也会一直处于高优先级，干扰 FLT 产生的流量。

我们通过数学建模的方式解决了如何寻找最优阈值的问题。为了方便阅读，我们将模型中使用了的部分参数集中放在了表 3.1 中。

3.2.1 大小包划分阈值

为了得到最优的划分阈值，我们根据 FLT 和 FHT 的数据包大小分布建立了一个最优化模型。s_p 表示数据包 p 的数据包大小。P_i 表示数据流 f_i 的数据包集合，即 $P_i = p_1, p_2, \cdots, p_{N_i}$，其中 N_i 表示数据流 f_i 的数据包数量。我们用 s_{min} 和 s_{max} 分别表示网络中数据包大小的最小值与最大值。

我们用 P_i^s 表示在 f_i 中数据包大小为 s 的数据包的集合，即

$$P_i^s = \{p | p \in P_i, s_p = s\} \tag{3-1}$$

所以 f_i 中大小为 s 的数据包所占的比例 $Pr_i(s)$ 为

$$Pr_i(s) = |P_i^s| / |P_i| \tag{3-2}$$

由于数据包大小分布是离散函数且最大值为 s_{max}，最小值为 s_{min}，所以对于 f_i，我们有

$$\sum_{k=s_{min}}^{s_{max}} Pr_i(k) = 1 \tag{3-3}$$

对于大小为 k 的数据包，我们并不直接使用 k 来更新计数器，而是使用一个关于数据包大小的映射函数 $g(\cdot)$，来更新该数据流的计数器。$g(\cdot)$ 关于的数据流 f_i 的定义为

$$g(f_i) = \sum_{k=s_{min}}^{s_{max}} N_i \cdot Pr_i(k) \cdot g(k) \tag{3-4}$$

假设区分大小数据包的阈值为 T。Panda 将一条数据流分成两部分，小包和大包，所以我们用 f_i^s 和 f_i^l 分别表示 f_i 中的小包集合与大包集合。因此，小包的流量占总流量的比例为

$$R_i^T = \frac{\sum_{k=s_{min}}^{T-1} Pr_i(k) \cdot k}{\sum_{k=s_{min}}^{s_{max}} Pr_i(k) \cdot k} \tag{3-5}$$

从公式（3-5）可以看出，R_i^T 只与数据包大小的分布有关，而与数据流中的数据包数量无关。如果 FLT 中的小包大小偏小时，公式（3-5）会导致 R_i^T 偏低，从而影响阈值 T 的选择。假如 $f \in FLT$，其数据包大小的分布为：大小为 100Bytes 的数据包占 95%，为 MTU 的数据包占 4%，为 10MTU 的占 1%，这三种大小的数据包的数据量比例为 95:60:150。小包所占的数量比例虽然为 95%，但其数据量却只占 ~31%，而只占 1% 的大包却占据了主要部分，这会导致数据流的数据包大小分布特性完全被掩盖。如果 Panda 选择的阈值 T 需要保证小包的数据量所占的比例尽可能大，那么有 $MTU \le T \le 10MTU$，但实际上应该是 $100 \le T \le MTU$。

为了解决因为FLT中小包大小偏小导致阈值 T 偏大的问题，我们改进了公式（3-5），不直接使用数据量这个评测指标，而是将数据包的大小映射成当数据包离开网卡时被切分后的数据包数量，即

$$R_i^T = \frac{\sum_{k=s_{min}}^{T-1} Pr_i(k) \cdot map(k)}{\sum_{k=s_{min}}^{s_{max}} Pr_i(k) \cdot map(k)} \tag{3-6}$$

其中，$map(k) = \lceil (k - header\_len)/MSS \rceil$，$header\_len$ 表示数据包包头的长度，$MSS + header\_len = MTU$。当小于等于MTU时，数据包大小被映射成1，这样便避免了小包过小的问题。我们仍使用数据流 f 的例子。使用公式（3-6）后，这三种大小的数据包的映射比例为95:4:10，大小为100Bytes的数据包所占的比例仍很突出。

建模的目的是为了找出最优的阈值来划分FLT与FHT。我们使用 L 和 H 分别表示FLT数据流和FHT数据流的集合，因此，我们有

$$\begin{aligned} L &= \{l_1, l_2, \cdots\} \\ H &= \{h_1, h_2, \cdots\} \end{aligned} \tag{3-7}$$

一个好的阈值应该要保证FLT的 R_i^T 尽可能大，而FHT的 R_i^T 尽可能小。但是，对于FLT和FHT，R_i^T 都是一个关于阈值 T 的单调递增函数。为了找到最优阈值，我们需要定义另一个关于阈值 T 的函数。Panda为每条数据流分配了一个权值（权值可以依据数据流中数据包的数量分配，也可以依据其他因素分配），w_i 表示 f_i 的权值。所以，对于一类数据流，我们有

$$R_C^T = \frac{\sum_{c \in C} w_c R_c^T}{\sum_{c \in C} w_c} \tag{3-8}$$

其中 $C \in \{L, H\}$。最优的阈值 T 应该要保证FLT一直会被分配最高优先级，而根据FHT已经发送的数据量为其分配优先级。但是，Panda在保护FLT产生的流量的同时，会不可避免地在一定程度上保护FHT产生的流量（无法保证FHT的数据包100%大于阈值）。因此，我们使用 R_L^T 与 R_H^T 的差值得到最优的阈值 T，最优化问题如公式（3-9）所示：

$$\max_{\{T\}} \qquad t(T) = \frac{\sum_{l \in L} w_l R_l^T}{\sum_{l \in L} w_l} - \frac{\sum_{h \in H} w_h R_h^T}{\sum_{h \in H} w_h} \tag{3-9}$$

其中，$T \in [s_{min}, s_{max}]$。当所有数据流的数据包大小分布已知时，$t(T)$ 是一个关于 T 的线性函数。

3.2.2 数据包大小映射函数

$g(\cdot)$ 的选择会直接影响 Panda 的性能。我们使用 $E_T(g(\cdot))$（我们称之为数据比）评测数据包大小映射函数的效果。$E_T(g(\cdot))$ 的定义为：当阈值为 T，被 Panda 降低到相同优先级时，FLT 发送的数据量与 FHT 发送的数据量的比值。例如，我们可以认为 PIAS 选择的数据包大小划分阈值为 0，即所有的数据包都是大包。由于 PIAS 选择的数据包大小映射函数为 $g(k)=k$，所以 PIAS 的数据比为

$$E_0(g(k))=1 \tag{3-10}$$

也就是说，PIAS 并没有区分 FLT 与 FHT，而是将这两种类型的数据流一视同仁。当使用公式（3-9）推导出最优阈值 T 后，即使仍然使用映射函数 $g(k)=k$，Panda 的数据比也会比 PIAS 好很多。比如，假设阈值 $T=1200$，一条 FLT 中大小为 1000 的数据包占 95%，而大小为 1500 的数据包占 5%；另一条 FHT 中大小为 1000 的数据包占 10%，而大小为 1500 的数据包占 90%。此时，Panda 的数据比为

$$E_{1200}(g(k)=k)=12.7 \tag{3-11}$$

远大于 PIAS 的数据比。但是，如果 FLT 中大部分都是很小的小包，如图 1.7 所示，此时 Panda 的数据比大约为 5。也就是说，Panda 的数据比与数据包的大小分布密切相关。

我们分析两种简单的数据包大小映射函数与数据包大小分布函数的关系。为了方便描述，我们使用 $avg_i^s(T)$，$avg_i^l(T)$，avg_i 分别表示当阈值大小为 T 时，f_i 的小包平均大小，大包平均大小和平均数据包大小；使用 $Pr_i^s(T)$ 与 $Pr_i^l(T)$ 表示小包与大包所占的比例，所以我们有 $Pr_i^s(T)+Pr_i^l(T)=1$。由于当数据流的数据包大小分布变化时，数据比也会发生变化，为了方便分析，我们给定两条参数数据流 $f_\alpha \in L$ 与 $f_\beta \in H$。f_l 的相关参数为 $avg_\alpha^s(T)$，$avg_\alpha^l(T)$，avg_α，$Pr_\alpha^s(T)$ 与 $Pr_\alpha^l(T)$，f_β 的相关参数为 $avg_\beta^s(T)$，$avg_\beta^l(T)$，avg_β，$Pr_\beta^s(T)$ 与 $Pr_\beta^l(T)$。

Case 1: 非补偿函数

$$g(k)=\begin{cases}k\,, k>T\\ 0\,, 0<k\le T\end{cases}$$

这种映射函数最为直观，当数据包大小大于阈值时，更新数据流的计数器，而当数据包小于阈值时，计数器不更新。为了求出数据比，我们假设当 f_α 与 f_β 被降到相同优先级时，它们已经发送的数据包的数量分别为 x 与 y。根据数据比的定义，我们可以得

到方程

$$\begin{cases} E_T(g(k)) = \frac{avg_\alpha^s(T)Pr_\alpha^s(T)x+avg_\alpha^l(T)Pr_\alpha^l(T)x}{avg_\beta^s(T)Pr_\beta^s(T)y+avg_\beta^l(T)Pr_\beta^l(T)y} \\ avg_\alpha^l(T)Pr_\alpha^l(T)\cdot x = avg_\beta^l(T)Pr_\beta^l(T)\cdot y \end{cases} \tag{3-12}$$

我们可以将方程（3-12）化简为:

$$E_T(g(k)) = \frac{avg_\alpha}{avg_\beta}\cdot\frac{avg_\beta^l(T)}{avg_\alpha^l(T)}\cdot\frac{Pr_\beta^l(T)}{Pr_\alpha^l(T)} \tag{3-13}$$

先讨论没有 offload 特性的网络环境。在公式（3-13）中，虽然 $Pr_\alpha^l(T)$，$Pr_\beta^l(T)$ 的值相差较大 [4]，但大包的平均大小与其所占的比例并无关系无关，$avg_\alpha^l(T)$ 与 $avg_\beta^l(T)$ 的值基本相同，均接近于 MTU。因此，公式（3-13）可进一步简化为：

$$E_T(g(k)) = \frac{avg_\alpha}{avg_\beta}\cdot\frac{Pr_\beta^l(T)}{Pr_\alpha^l(T)} < \frac{Pr_\beta^l(T)}{Pr_\alpha^l(T)} \tag{3-14}$$

由于 $avg_\alpha < avg_\beta$，所以我们得到

$$E_T(g(k)) < \frac{Pr_\beta^l(T)}{Pr_\alpha^l(T)} \tag{3-15}$$

而在开启了 offload 特性的网络环境中，FLT 与 FHT 的数据包大小分布差异更加明显，如图 1.8(b) 所示。虽然根据公式（3-9）推导出的最优阈值还是接近于 MTU，但此时 $avg_\beta^l(T)$ 却是明显大于 $avg_\alpha^l(T)$（由于 FHT 中的数据包大小相比于 FLT 更趋近于 64KB），所以此时的数据比会更大一些，也就是说，Panda 的性能会更好。

但如果 FLT 中的数据包绝大部分都是极小的包，这会减小 FLT 的平均数据包大小，从而降低 Panda 的数据比，如在如图 1.8(a) 所示的数据包分布下，Panda 的数据比只有 5 左右。为了进一步提高 Panda 的数据比，我们设计了另一种数据包大小映射函数。

Case 2: 补偿函数

$$g(k) = \begin{cases} k\ , k > T \\ k - T\ , 0 < k \le T \end{cases}$$

当数据包大于阈值时，说明该数据流更有可能是 FHT，从而增大其计数器的值；而当数据包小于阈值时，该数据流更有可能是 FLT，所以对计数器进行补偿。数据包越小，对计数器的补偿越大。同Case 1一样，我们假设当 f_α 与 f_β 被降到相同优先级时，它们已经发送的数据包的数量分别为 x 与 y。根据数据比的定义，我们可以得到方程

$$\begin{cases} E_T(g(k)) = (avg_\alpha^s(T)Pr_\alpha^s(T)x + avg_\alpha^l(T)Pr_\alpha^l(T)x)/(avg_\beta^s(T)Pr_\beta^s(T)y + avg_\beta^l(T)Pr_\beta^l(T)y) \\ avg_\alpha^l(T)Pr_\alpha^l(T)x - \sum_{k=s_{min}}^{T}(T-k)Pr_\alpha(k)x = avg_\beta^l(T)Pr_\beta^l(T)y - \sum_{k=s_{min}}^{T}(T-k)Pr_\beta(k)y \end{cases} \tag{3-16}$$

由于

$$\begin{aligned} &\sum_{k=s_{min}}^{T}(T-k)Pr_\alpha(k) \\ =&T\sum_{k=s_{min}}^{T}Pr_\alpha(k) - \sum_{k=s_{min}}^{T}kPr_\alpha(k) \\ =&TPr_\alpha^s(T) - avg_\alpha^s(T)Pr_\alpha^s(T) \end{aligned} \tag{3-17}$$

将方程（3-17）带入（3-16）得到：

$$\begin{cases} E_T(g(k)) = avg_\alpha x/avg_\beta y \\ (avg_\alpha - TPr_\alpha^s(T))x = (avg_\beta - TPr_\beta^s(T))y \end{cases} \tag{3-18}$$

注意到，对于 FHT，小包通常只占很小的比例（$Pr_\beta^s(T)$ 一般小于 5%），所以 $avg_\beta > T$，因此

$$avg_\beta - TPr_\beta^s(T) > 0 \tag{3-19}$$

但小包在 FLT 中所占比例很大，很有可能出现 $avg_\alpha - TPr_\alpha^s(T) \le 0$ 的情形。当这种情况出现时，FLT 就会一直处于最高优先级。而当 $avg_\alpha - TPr_\alpha^s(T) > 0$ 时，我们可以得到：

$$\begin{aligned} E_T(g(k)) &= \frac{avg_\alpha}{avg_\beta}\cdot\frac{avg_\beta - TPr_\beta^s(T)}{avg_\alpha - TPr_\alpha^s(T)} \\ &\approx \frac{avg_\alpha}{avg_\beta}\cdot\frac{avg_\beta}{avg_\alpha - TPr_\alpha^s(T)} \\ &= \frac{avg_\alpha}{avg_\alpha - TPr_\alpha^s(T)} \end{aligned} \tag{3-20}$$

$E_T(g(k))$ 是一个关于 avg_α 的单调递减函数，所以 FLT 中小包的数据包大小越小，Panda 的数据比越大。而即使当 FLT 中小包的数据包大小偏大时，即 avg_α 趋近于 T 时，Panda 的数据比为

$$E_T(g(k)) \ge \frac{1}{1 - Pr_\alpha^s(T)} = \frac{1}{Pr_\alpha^l(T)} \tag{3-21}$$

与公式（3-15）比较，Panda 使用补偿函数的数据比要明显好于非补偿函数。当使用补偿函数时，Panda 的数据比范围为 $(1/Pr_\alpha^l(T), +\infty)$，且小包的数据包大小越小，数据比越高。

当使用非补偿函数时，数据包大小出现的序列会影响 Panda 的数据比。举个例子。如果一条 FLT 数据流中先产生的一直是大包，Panda 会增加其计数器的值，当该值大于某个优先级的阈值时，Panda 会将该数据流降低至低优先级。即使之后数据流产生的数据包全是小包，这些小包也不得不进入低优先级对应的队列中等待调度，导致其响应延时增加。也就是说，Panda 的数据比会显著下降。而当使用补偿函数时，数据包大小出现的序列会基本不会影响 Panda 的数据比。即使数据流先产生的数据包均为大包，导致其被降低至某个低优先级，但由于之后的数据包都是小包，这会对其计数器进行补偿。一段时间后，Panda 会提升该数据流的优先级，从而保证之后产生的小包的低延时。因此，Panda 的数据比并不会被明显降低。

3.2.3 在 offload 特性下区分 FHT

我们根据数据流的大小将吞吐密集型应用产生的数据流分成了三类：小流、中流和大流（§1.3.1）。如图 1.8(a) 所示，在没有 offload 特性的网络环境下，这三种数据流产生的数据包大小分布函数没有明显区别，超过 95% 的数据包大小都为 MTU。而当使用了 offload 特性时，如图 1.8(b) 所示，三种数据流的数据包大小分布函数区别明显，因此，我们根据这个区别进一步区分小流、中流和大流，在不需要预知数据流大小的前提下，优化平均流完成时间。

对于一条 FHT 数据流，如果其产生的数据包大小较小（比如大多数都小于等于 6MTU），那么，这条流更有可能是小流；而如果其产生的数据包大部分都是接近于 64KB 的数据包大小，那么这条流更有可能是大流。如果我们能预知数据流大小，那么我们可以根据该信息直接为数据流分配优先级：数据流大小越大，其优先级越低。但由于数据流大小未知，Panda 无法在数据流开始传输时就确定其最终的优先级。但由于具有不同数据流大小的数据流的数据包大小分布差异很大，我们可以根据这一信息让大流更快地落入低优先级。Panda 采用的方案是对大数据包进行惩罚。

在选定了区分大小包的阈值 T 后，Panda 在 $(T, s_{max}]$ 内继续找到 n 个数据包划分阈值，分别记为 $t_1, t_2, \cdots, t_n$。为了方便描述，我们令 $t_0 = T$，所以有 $t_0 < t_1 < t_2 < \cdots < t_n$。为了对大数据包进行惩罚，Panda 为大小处于不同区间内的数据包设定了不同权值。我们令 w^i 为区间 $(t_i, t_{i+1}]$ 的权值，其中 $i \in [0, n-1]$，且 $1 = w^0 \leq w^1 \leq \cdots \leq w^{n-1}$。

Panda 更新 counter 的算法如算法2所示。当一个数据包产生时，如果其数据包大小 $s_p < T$，Panda 使用补偿函数减小该数据流的计数器；而当 $s_p > T$ 时，Panda 首先获取到该数据包大小所处的区间，得到对应的权值，然后将计数器增加 $s_p \cdot w_i$；最后返回计数器的值所对应的优先级。

如何确定这些阈值是我们接下来要解决的问题。我们使用类似公式（3-9）的方法得到这些阈值。

求解阈值 Panda 先收集历史信息，将 FHT 数据流根据其数据流大小划分成 m 类，划

Algorithm 2 Panda 更新算法

Input: pkt_size

Output: priority

```
for each packet generates do
    if pkt_size < T then
        counter− = T − pkt_size
    else
        w = get_weight(pkt_size)
        counter+ = w · pkt_size
    end if
    return get_priority(counter)
end for
```

分阈值分别为 $d_1, d_2, \cdots, d_{m-1}$。为了描述方便，令 $d_0 = 0, d_m = \infty$。我们使用 $C_1, C_2, \cdots, C_m$ 分别表示这 m 类数据流的集合。对于任意一条数据流 $f \in C_i$，其中 $i \in [1, m]$，有 $|f| \in [d_{i-1}, d_i)$，$|f|$ 表示数据流 f 的大小。

为了找到划分 C_i 与 C_{i+1} 的数据包大小阈值 d_i，其中 $i \in [1, m-1]$，根据公式（3-6），当 $f_i \in C_i \bigcup C_{i+1}$ 时，我们同样定义

$$R_i^{d_i} = \frac{\sum_{k=s_{min}}^{d_i-1} Pr_i(k) \cdot map(k)}{\sum_{k=s_{min}}^{s_{max}} Pr_i(k) \cdot map(k)} \tag{3-22}$$

根据公式（3-9），我们可以得到求解 d_i 的最优化问题：

$$\max_{\{d_i\}} \quad t(d_i) = \frac{\sum_{f \in C_i} w_f R_f^{d_i}}{\sum_{f \in C_i} w_f} - \frac{\sum_{f \in C_{i+1}} w_f R_f^{d_i}}{\sum_{f \in C_{i+1}} w_f} \tag{3-23}$$

其中，$T \in [s_{min}, s_{max}]$，w_f 表示数据流 f 的权值。

3.2.4 划分优先级的阈值

Panda 更新数据流的计数器的目的，是为了根据计数器的值估计数据流可能的类型，从而为数据流分配合适的优先级，保证这些数据流的服务质量。只有当计数器的值超过一个阈值后，优先级才开始降低。因此，如何选择这些用来划分优先级的阈值对 Panda 的性能影响很大。

问题建模 假设系统中需要 K 个优先级队列 $P_i (1 \le i \le K)$，P_1 表示最高优先级队列。我们用 $\alpha_{j-1} (2 \le j \le K)$ 表示数据流从优先级 $j-1$ 降到优先级 j 的阈值。为了方便描述，令 $\alpha_K = x_{max}$，其中 x_{max} 表示数据流大小的最大值，这样可能保证最大的数据流也能一直处于这些队列中，令 $\alpha_0 = 0$。假设共有 M 条 FHT 数据流样本，我们对这些数据流从 $i = 1$ 到 $i = M$ 进行编号，f_i 表示编号为 i 的 FHT 数据流。由于优先级的划分阈值主要

针对 FHT 数据流而设计，而在这类数据流中，小包的比例很低，所以我们忽略小包的影响，只考虑大包。

对于任意一条 FHT 数据流 f_i，阈值 $d_1, d_2, \cdots, d_{m-1}$ 能将其划分成 m 个部分，我们假设第 k 部分 $(1 \le k \le m)$ 的数据量为 q_i^k，通过算法 2（忽略小于阈值 T 的数据包），我们能得到一个与该条数据流对应的映射值，记为 x_i，所以我们有

$$\begin{aligned} x_i &= \sum_{k=0}^{m-1} w^k q_i^k \\ |f_i| &= \sum_{k=0}^{m-1} q_i^k \end{aligned} \tag{3-24}$$

其中 $i \in [1, M]$。$F(x)$ 表示数据流通过算法 2 映射后的累计密度函数，所以 $F(x)$ 表示映射不超过 x 的数据流所占的比例。值得注意的是，我们并没有对 $F(x)$ 做任何假设。

令 $\theta_j = F(\alpha_j) - F(\alpha_{j-1})$，表示映射大小在区间 $[\alpha_{j-1}, \alpha_j)$ 内的数据流所占的比例。对于一条映射值处于区间 $[\alpha_{j-1}, \alpha_j)$ 内的数据流 f_i，它需要经历从队列 P_1 到 P_j 的排队延时，且需要在 $P_k(1 \le k \le j-1)$ 内传输的平均数据量为 $\eta(\alpha_j - \alpha_{j-1})$，在 P_j 中的平均传输量为 $x^+ = \eta(x_i - \alpha_{max}(x_i))$，其中 η 表示映射系数，其值为 $\eta = |f_i|/x_i$，$\alpha_{max}(x_i)$ 表示映射值为 x_i 的数据流被降低到最低优先级的阈值，$j_{max}(x_i)$ 表示该阈值对应的编号。

我们使用 T_j 表示单位大小的数据包在 P_j 中所经历的平均延时，因此，映射值为 x 的数据流的平均流完成时间为

$$T(x) = \sum_{l=1}^{j_{max}(x)} \eta(\alpha_l - \alpha_{l-1}) T_l + x^+ T_{j_{max}(x)+1} \tag{3-25}$$

对于属于同一类的数据流（$C_1, C_2, \cdots, C_m$ 中的一类），它们的映射系数基本相同，但对于不同类的数据流，映射系统可能差别很大。因此，我们用 η_i 表示 $C_i(1 \le i \le m)$ 的平均映射系数，p_i 表示属于 C_i 的数据流的数量占全部 FHT 数据流的比例。

为了简化分析，我们假设系统中数据流的映射值均匀分布。对于映射值在区间 $[\alpha_{j-1}, \alpha_j)$ 内的数据流，它们的平均流完成时间为

$$\begin{aligned} T_j &= \sum_{l=1}^{j-1} \sum_{c=1}^{m} p_c \eta_c (\alpha_l - \alpha_{l-1}) T_l + \sum_{c=1}^{m} \frac{1}{2} p_c \eta_c (\alpha_j - \alpha_{j-1}) T_j \\ &= (\sum_{j=1}^{m} p_j \eta_j) \sum_{i=1}^{j-1} (\alpha_i - \alpha_{i-1}) T_i + (\frac{1}{2} \sum_{c=1}^{m} p_c \eta_c)(\alpha_j - \alpha_{j-1}) T_j \end{aligned} \tag{3-26}$$

令 $Q = \sum_{c=1}^{m} p_j \eta_j$，公式（3-26）可以化简为

$$T_j = Q \sum_{i=1}^{j-1} (\alpha_i - \alpha_{i-1}) T_i + \frac{1}{2} Q (\alpha_j - \alpha_{j-1}) T_j \tag{3-27}$$

因此，数据流在某条平均链路的平均流完成时间为

$$\begin{aligned} T &= \sum_{j=1}^{K} \theta_j T_j \\ &= Q \sum_{j=1}^{K} \theta_j \left(\sum_{i=1}^{j-1} (\alpha_i - \alpha_{i-1}) T_i + \frac{1}{2} (\alpha_j - \alpha_{j-1}) T_j \right) \\ &= Q \sum_{j=1}^{K} \theta_j \sum_{i=1}^{j-1} (\alpha_i - \alpha_{i-1}) T_i + Q \sum_{j=1}^{K} \frac{1}{2} \theta_j (\alpha_j - \alpha_{j-1}) T_j \end{aligned} \tag{3-28}$$

为了方便分析，我们记 $U_i = (\alpha_i - \alpha_{i-1}) T_i$，其中 $i \in [1, K]$。公式（3-28）可以转换为

$$T = Q \sum_{j=1}^{K} \theta_j \sum_{i=1}^{j-1} U_i + Q \sum_{j=1}^{K} \frac{1}{2} \theta_j U_j \tag{3-29}$$

因此，求解最优的优先级阈值 $\{\alpha_j\}$ 的问题转换成了一个最小化问题：

$$\begin{aligned} \min_{\{\alpha_i\}} \quad & T = \sum_{j=1}^{K} \theta_j \sum_{i=1}^{j-1} U_i + \sum_{j=1}^{K} \frac{1}{2} \theta_j U_j \\ \text{suject to} \quad & \alpha_0 = 0, \alpha_K = x_{max} \\ & \alpha_{j-1} < \alpha_j, j = 1, 2, \cdots, K \end{aligned} \tag{3-30}$$

[12] 给出了求解该最小化问题的近似算法（见附录 B）。

3.3 在线分类

离线建模模块使用收集的历史数据得到了划分大小包的最优阈值 T，保证 FLT 产生的流量尽可能长时间地处于高优先级；得到划分各类 FHT 和优先级的阈值，优化 FHT 的平均流完成时间。之后，Panda 将这些参数应用到实际系统中来同时满足 FLT 和 FHT 的服务质量要求。Panda 的在线分类过程由三部分组成：Panda 标记数据包优先级，Linux TC 多优先级队列调度，交换机多优先级队列调度，如图 3.1 所示。接下来分别介绍这三个部分。

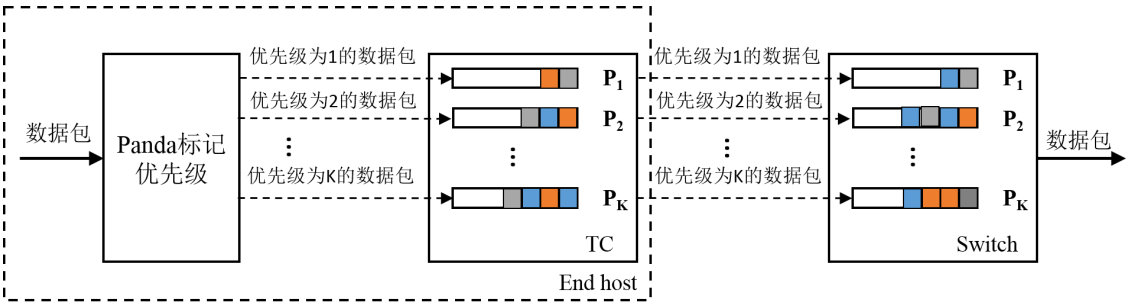


图 3.2 Panda 在线分类过程

标记优先级 Panda 分布式地部署在每一台终端中，通过实现一个内核模块来更新数据流的优先级。一个数据包产生后，Panda 得到其数据包大小，然后通过算法2更新其计数器。之后 Panda 根据计数器的值与优先级阈值进行比较，返回数据流的优先级，并将该优先级标记在 DSCP 字段上，方便 AQM 与交换机识别。数据包在进入 Linux TC 前被标记上优先级。

终端配置 Panda 需要终端的多优先级队列 AQM 的支持。数据包被打上优先级标签后，在它们在离开终端前，会先进入 TC 等待调度。我们的测试结果 (§1.1) 表明，如果没有合适的调度策略，延时敏感型应用产生的数据包会在 TC 中经历非常严重的排队延时。Panda 配合一个多优先级队列 AQM 可以有效解决终端的排队延时问题。当被标记了优先级的数据包到达 TC 后，它们根据其优先级进入对应的队列中。由于 FLT 一直处于高优先级，所以 Panda 可以保证 FLT 产生的流量在终端的低延时；此外，由于 FHT 的数据流大小越大，优先级越低，所以也可以优化 FHT 在终端的平均流完成时间。

交换机配置 Panda 需要商用交换机中多优先级队列的支持。数据包进入交换机后，会根据它们自身的优先级进入对应的优先级队列中，从而保证了 FLT 产生的流量在交换机中的低延时，也优化了 FHT 在交换机的平均流完成时间。

速率控制 Panda 推荐使用 DCTCP[5] 作为终端的传输层协议。由于现有的商用交换机普遍支持 ECN 打标，所以，只需要将交换机设置成支持 ECN 即可。在终端，Panda 也可以通过设置 ECN 打标降低队列长度。在没有 offload 特性的网络中，数据包大小最大为 MTU。因此，我们可以设置一个阈值，当队列长度超过该阈值时，对数据包进行打标。但在开启了 offload 特性的网络中，数据包最大为 64KB，此时对数据包打标会遇到两个问题。

第一个问题是阈值的设置。[5] 中建议的阈值配置是 10Gbps 网络中推荐为 100KB，1Gbps 网络中为 30KB，但该配置的对象是交换机，也就是说，打标的数据包大小最大仍为 MTU。但在终端进行打标时，最大数据包大小为 64KB。如果仍然使用推荐值的话，会导致大量的数据包都会被打上 CE 标记。如果我们把阈值设置为 100KB，当队列中同时存在两个 64KB 的数据包时，就已经超过了阈值，也就是说，第二个数据包会被打上标记。但是如果我们为了改善这种情形而增大阈值的话，则会增大终端排队延时。

第二个问题是 ECN 标记比例过大的问题。当一个大数据包在终端被打上 ECN 标记后，网卡在将这个大数据包分成大小为 MTU 的数据包时，每个数据包都会被打上 ECN 标记，但这会导致 ECN 标记比例偏大。假设 ECN 打标阈值还是 100KB，第一个 64KB 的数据包已经在 TC 的队列中等待调度。当第二个 64KB 的数据包到达 TC 时，由于排队长度已经超过了 100KB 的阈值，所以第二个数据包会被打上 ECN 标记。而当该数据包离开网卡时，它会被分成多个大小为 MTU 的数据包，而且每个数据包都被标记了 CE（congestion experienced）。但实际上，只应该有 $100KB - 64KB = 36KB$ 的数据包被打上标记，但现在被打上 ECN 标记的数据包为 64KB，增加了 78%。

以上两个问题都会导致 ECN 打标比例过大，使得拥塞窗口大小减小过快，最终可能会导致系统无法充分利用网络带宽，影响性能。

为了避免这个问题，在开启了 offload 特性的网络中，Panda 暂时没有在终端使用 ECN 打标的策略。由于 FLT 一直处于高优先级，这对 FLT 产生的流量的排队延时影响不大，但由于 FHT 产生的数据包会在低优先级队列排队，如果没有通过 ECN 标记的方式限制队列长度的话，会增加数据包的排队延时，导致平均流完成时间增大。

3.4 小结

本章详细介绍了 Panda 的三个组成部分：数据收集模块，离线建模模块与在线分类模块。数据收集模块收集数据中心内的历史数据；离线建模模块通过历史数据和数学模型得到最优参数；而在线分类模块则将最优参数应用在系统中，为数据流分配优先级。离线建模模块主要解决了三个问题：1）选取划分大小数据包的最优阈值；2）根据数据包大小更新数据流的计数器；3）根据计数器的值分配优先级。在线分类模块则需要耦合数据包的优先级与多优先级队列，才能保证延时敏感型应用的低延时和优化吞吐密集型应用的平均流完成时间。我们已经有了较为完善的数学模型，接下来则需要在真实系统中实现 Panda 调度算法。

第四章　Panda 的实现与实验环境

我们在 Linux 平台上实现了一个 Panda 的原型系统。原型系统主要包括两部分：数据包优先级标记和多优先级队列。

4.1　数据包优先级标记

数据包优先级标记模块在终端实现，负责维护每条数据流的状态（主要是其计数器的值）和在数据包的 DSCP 字段标记上数据流的优先级。我们在 Linux 平台上将这个模块实现成了一个 kernel module，可实时安装与卸载，方便使用。这个 module 位于 TCP/IP 协议栈与 Linux TC 之间，如图 4.1 所示。当数据包离开 TCP/IP 协议栈后，会进入该 module 进行处理，更新计数器的值，标记优先级，之后进入 TC 等待调度。这个 module 包括三个部分：一个 NETFILTER[48] 钩子（hook），一个基于哈希算法的流表（flow table）和一个数据包修改器。

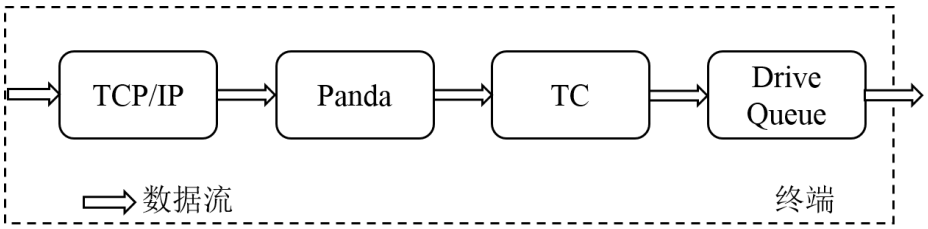


图 4.1　数据包标记模块所在的位置

数据包的标记流程如下：1）NETFILTER 通过 LOCAL_OUT 钩子拦截所有即将离开终端的数据包（在进入 TC 之前）。2）每条数据流通过源 IP、目的 IP、源端口、目的端口四元组唯一识别。当一个数据包进入 module 后，我们通过数据包的四元组找出其对应的数据流，而当该数据流不在流表中时，新建一个表项。Panda 获得数据包的大小后，根据更新算法更新这条数据流的计数器大小。3）根据计数器的大小，Panda 为该数据流分配一个优先级，然后将该优先级通过数据包修改器填写在数据包的 DSCP 字段。

数据包被钩子拦截后的处理流程如图 4.2 所示。1）如果数据包使用的不是 IP 协议，则不做处理，直接返回。2）之后判断数据包是否使用 TCP 协议，如果不是，将该数据包的优先级设置为 1（最优优先级）。3）否则判断该数据包是否为 SYN 包。如果是，说明这是一条新建立的数据流，需要在流表中新建一条与该数据流对应的表项。4）如果不是 SYN 包而是 FIN 包或者 RST 包，说明这条数据流已经关闭，与该数据流对应的表项可以从流表中删除了。SYN、FIN 和 RST 这三类数据包表明的是连接的状态，需要优先服务，所以它们的优先级都被设置成 1。5）如果是一般数据包，我们先从数据包的头部信息中获取到 payload 的长度，并通过四元组匹配到流表中与之对应的表项。6）然后

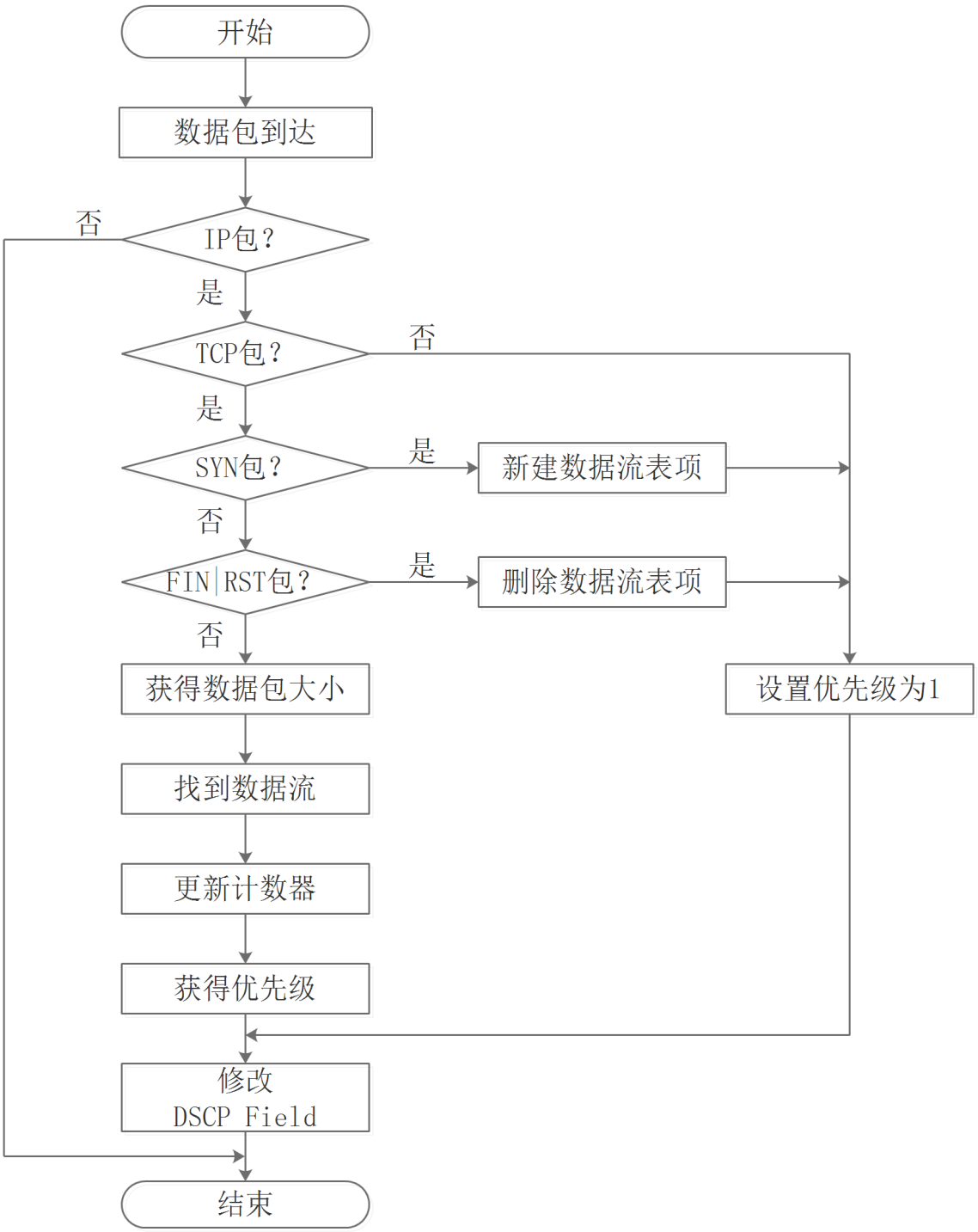


图 4.2　钩子拦截数据包后的处理流程

根据更新算法更新该数据流的计数器。7）获取到与计数器的值对应的优先级后，数据包修改器在数据包的 IP 头 DSCP 字段填入优先级的值。8）返回。

4.2 多优先级队列

我们在 Linux 平台上实现了一个多优先级队列 AQM，其严格按照队列的优先级顺序对数据包进行调度。数据包到达 TC 后，其会根据 DSCP 字段进入与其优先级对应的队列中。由于 Panda 推荐使用 DCTCP 作为终端的传输层协议，因此，我们可以通过在 AQM 里实现 ECN 打标的方式控制数据包在 TC 的队列长度，降低排队延时。但 ECN 标记在多优先级队列中的使用存在一个问题，即打标阈值的设定是以队列为粒度还是以端口为粒度 [49]。ECN 标记在单队列方案中不会遇到这个问题。在设定好单队列的 ECN 标记阈值后，当队列长度超过该阈值时，数据包便会被打上 CE 标志。但在多优先级队列中，阈值的设定却并不简单。假设 ECN 标记阈值为 T，系统中有 e 个优先级队列。

方案一 如果每个队列都使用 T 作为 ECN 标记阈值，虽然具有最高优先级的数据包能得到很好的服务，但优先级较低的数据包则不得不经受很大的排队延时。由于多优先级队列严格按照优先级顺序进行调度，当高优先级队列有数据包时，在低优先级队列等待的数据包将不会被服务。那么，优先级为 $i(1 \le i \le e)$ 的队列的平均队列长度约为 iT。由于 FLT 产生的流量总是处于高优先级，所以这种方式并不会影响 FLT 的低延时服务质量要求。但对于 FHT，它们产生的流量会进入低优先级队列中，导致数据包的平均 RTT 大大延长，从而大幅增加了 FHT 的平均流完成时间。

方案二 所有队列平分阈值。虽然这种方案能保证处于低优先级队列的数据包的低排队延时，但它有可能导致网络带宽不能被充分利用。每条队列的 ECN 标记阈值为 T/e，当 T 为 30KB，优先级队列数量为 8 时，每个队列的标记阈值约为 4KB，不到三个 MTU。在一段时间内，网络中产生的流量很有可能都是大流，所以 Panda 会给它们同时分配较低优先级。也就是说，虽然有多个优先级队列，但实际使用的队列数量却很少，甚至只有一个。T/e 的 ECN 标记阈值无法保证网络带宽被充分利用，导致网络性能降低。

方案三 所有队列共享阈值，也就是阈值的设置以端口为粒度。当所有队列的队列长度总和超过阈值时，进入任一队列的数据包都会被标记 CE。在这种方案下，优先级最低的数据包会经历的队列长度被限制为 T；而当网络中都是大流时，由于 ECN 标记阈值仍为 T，所以网络带宽也可以被充分利用。

因此，我们选择方案三作为交换机和终端的 ECN 标记策略。

4.3 CPU 和内存开销

Panda 为每条 TCP 数据流分配一个计数器，用于记录数据流的状态和其对应的优先级信息。随着虚拟化技术和容器技术的发展，单台物理机内产生的并发连接数可能高达数万条，甚至几十万条，因此，我们需要评估系统使用 Panda 作为流调度算法时的 CPU 和内存开销。

Panda 使用如图 4.3 所示的数据结构存储数据流信息。当数据包到达时，Panda 使

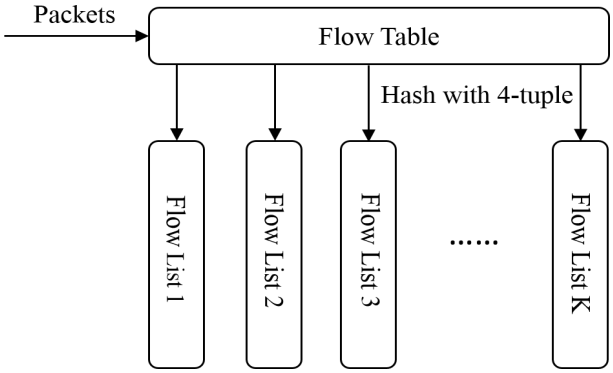


图 4.3　Panda 使用的数据结构

用数据包的四元组信息得到一个哈希值，并在该哈希值对应的链表中遍历查询该数据包对应的数据流信息。如果该数据流存在，则更新数据流对应的计数器值，如果不存在，则新建一条与该数据流对应的表项。为了减少哈希冲突，我们目前使用的链表数量为4096。我们使用四元组唯一标识一条数据流，加上计数器的开销，每条数据流需要的存储空间不到 30 Bytes。因此，即使单台物理机内产生的并发连接数高达 100 万条，Panda 所需要的内存开销仍不足 30 MBytes。此外，我们使用 5000 条并发数据流测试 Panda 的 CPU 开销。使用 Panda 时物理机的 CPU 使用率仅比不使用 Panda 时高出不到 2%，这说明 Panda 的 CPU 开销很小。

4.4　实验环境配置

我们分别在 1Gbps 和 10Gbps 网络环境下对 Panda 的性能进行了测试。在 1Gbps 网络中，我们同时关闭了发送端和接收端的 TSO 和 GRO；而在 10Gbps 网络中，为了充分利用带宽，我们同时开启了发送端和接收端的 TSO 和 GRO。由于我们主要评测数据包在终端产生的排队延时，为了消除交换机的影响，我们将两台物理机直连，如图 4.4 所示。

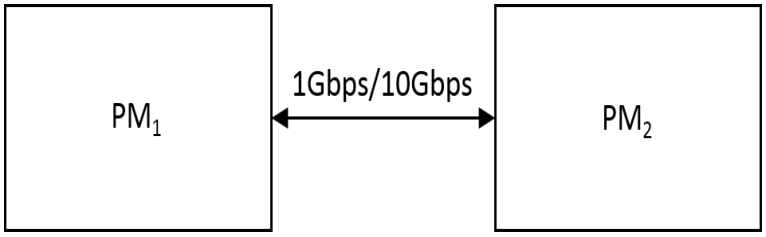


图 4.4　Panda 实验环境拓扑

实验平台　我们使用的 Linux 内核版本为 3.19.3[50]，该版本可以直接选择 DCTCP 作为传输层协议，而不需要另外安装补丁，操作系统版本为 Centos 6.5。使用的两台物理机的型号均为 Huawei Tecal RH2285 服务器，包含 6 个 Intel Xeon 2.40GHz 的 CPU，32GB 的内存，1Gbps 环境下网卡的型号为 Intel 82580 Gigabit，10Gbps 环境下网卡的型号为 Intel 82599ES 10-Gigabit。两台物理机直连后，基准 RTT（the base RTT）大约为 0.15ms。

参数设置 在 1Gbps 网络中，ECN 标记阈值为 30KB；在 10Gbps 网络中，由于存在 §3.3 中讨论的问题，所以我们在终端未使用 ECN 标记。多优先级队列的队列数量为 8。数据包在离开网卡前，会在网卡上形成一定的队列长度，增加排队延时。为了降低数据包在网卡排队产生的影响，我们将 1Gbps 网络下的 BQL（Byte Queue Limits）设为 10KB，10Gbps 网络下设为 128KB。由于本论文并没有解决在划分 FHT 类别后的权值设置，所以本论文暂时将所有权值设为相同。在这种情况下，Panda 与 PIAS 在优化 FHT 平均流完成时间上的性能基本一致。

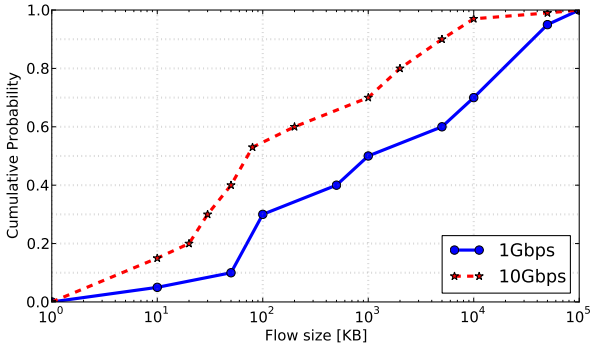


图 4.5 网络搜索流量负载的数据流大小 CDF

流量负载 我们使用一组真实的网页搜索流量负载 [5] 来模拟实验中的 FHT，其数据流大小分布如图 4.5 所示 [12]。我们将 FHT 的数据流分成三组：小流 $(0, 100KB]$, 中流 $(100KB, 10MB]$ 和大流 $(10MB, \infty)$。另外，我们使用 Memcached[24] 作为实验中的 FLT。我们建立一条 Memcached 的 TCP 长连接，其产生的 Key-Value 对中，所有的 key 值大小均为 64KB，而其 value 的大小分布如图 4.6 所示，该分布与图 1.7 中 FLT 数据包大小分布一致。在 1Gbps 网络下，只有大约 5% 的 value 大小为 MTU，而在 10Gbps 网络，大约有 3% 的 value 大于等于 MTU，而最大的 value 大小可以达到 15KB。

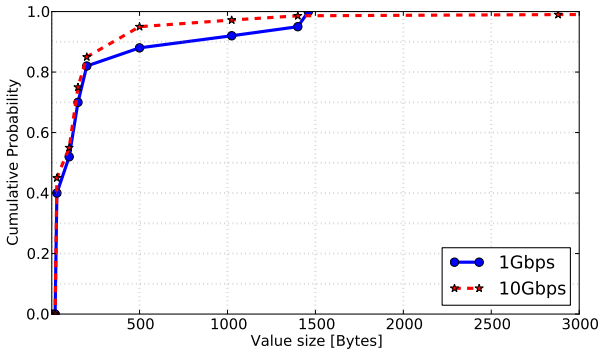


图 4.6 Memcached 中 value 大小分布 CDF

性能指标 我们使用 FLT 产生的请求的响应延时和 FHT 的平均流完成时间两个指标评测流调度算法的性能。FLT 需要的服务质量保证是低延时，而 FHT 需要的服务质

量保证是高吞吐和最小化平均流完成时间。由于本论文主要解决的问题是如何保证延时敏感型应用的低延时和优化吞吐密集型应用的平均流完成时间，因此，我们选用这两个指标评测一中流调度算法是否能同时满足这两种数据流的服务质量要求。

比较对象 在相关工作（§2）里，我们讨论过的算法包括 RED、CoDel、DCTCP、QJUMP 和 PIAS。由于 DCTCP 的性能要优于 RED 和 CoDel[5]，所以 Panda 只与 DCTCP 进行比较，而不与 RED 和 CoDel 进行比较。QJUMP 需要提前预知应用的优先级信息才能工作，但 Panda 不假设数据流信息已知，所以，Panda 也不与 QJUMP 进行比较。因此，在性能评测中，Panda 的比较对象包括 DCTCP 和 PIAS。

测试方法 为了产生如图 4.5 所示的数据流大小，我们使用 Epoll[51] 框架实现了一个负载发生器与一个负载服务器。该负载发生器可以通过设置数据流大小分布模拟真实的流量负载，如数据挖掘流量负载和网页搜索流量负载等。负载发生器的作用是向负载服务器发送请求，要求服务器返回指定大小的数据块。我们可以通过设置数据流的并发度，即同时向负载服务器请求数据的数据流的数量，来改变网络的拥塞程度。可以通过https://github.com/alvenwong/generator获取到源代码。

如图 4.4 所示，我们将 PM_1 作为负载发生器，PM_2 作为负载服务器。所以，数据包会在 PM_2 的 TC 处产生排队。另外，对于 Memcached，我们同样使用 PM_1 作为客户端，PM_2 作为服务器端。客户端向服务器端发送请求，要求服务器端返回指定大小的 value。FHT 产生的流量会在 PM_2 的 TC 处干扰 FLT 产生的流量。所以，我们在 PM_2 上部署了 Panda，保证 FLT 的低延时，同时优化 FHT 的平均流完成时间。

为了避免实验结果的偶然性，我们对每组实验重复了 10 次，去掉最大值和最小值后，取剩下的 8 组数据的平均值作为这组实验的实验结果。

4.5 小结

本章主要描述了 Panda 在 Linux 平台上的实现：使用 NETFILTER 机制钩住即将离开终端的数据包；得到数据包大小后，更新数据流对应的计数器；并将计数器值对应的优先级标记在数据包的 DSCP 域。在多优先级队列中，Panda 以端口为粒度设置 ECN 标记阈值来保证系统的性能。本章还介绍了评测 Panda 性能的实验环境和流量负载。为了消除交换机排队延时的影响，我们将两台物理机直连。我们用 Memcached 与真实的网络搜索流量负载分别作为延时敏感型应用与吞吐密集型应用。有了实验平台后，我们便可以测试 Panda 的性能。

第五章　性能评测

5.1　延时敏感型应用的低延时保证

我们分别测试了 Panda、PIAS 与 DCTCP 的性能，即在同样的 FHT 流量干扰下，FLT 的请求延时。我们建立一条 Memcached 的 TCP 连接。由于已经确定了请求的 value 大小的分布，所以，我们通过改变连接中请求的数量调整 FLT 数据流的大小。请求的数量范围为 100~10000。

5.1.1　1Gbps 网络环境测试

5.1.1.1　与数据流大小的关系

在 1Gbps 网络中，Panda 与 PIAS 使用 CUBIC 作为传输层协议时的实验结果如图 5.1 与图 5.2 所示。实验中的 FHT 的数据流并发度为 14，即同一时刻系统中同时发送数据的数据流的数量为 14，此时的流量负载大约为 0.8。从这两个图中我们可以得到如下两个结论。

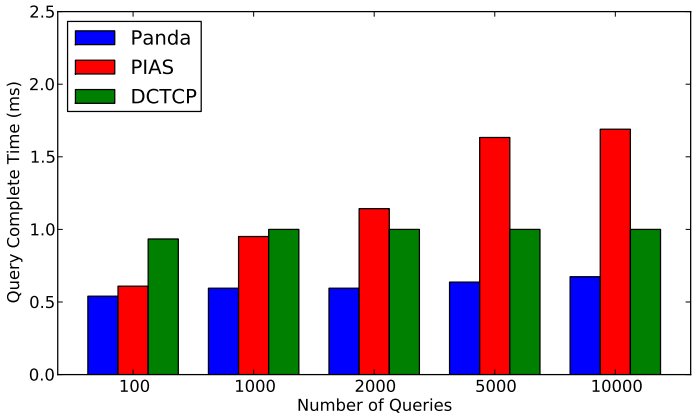


图 5.1　使用 CUBIC 时 Memcached 的请求延时 90% 分位值比较

首先，当使用 Panda 时，Memcached 的请求延时最低。在请求量只有 100 个时，由于 FLT 数据流大小较小，所以 PIAS 给 Memcached 分配较高优先级，此时 PIAS 与 Panda 的性能比较接近。这部分请求延时基本来自于在最高优先级队列中的排队延时，大约为 0.6ms。由于产生的负载中，小流仍占有部分比例，它们产生的数据包会在最高优先级产生队列，所以这部分排队延时无法避免。而随着请求量的增大，FLT 数据流大小也随之增加，导致 PIAS 会逐渐降低 Memcached 数据流的优先级。因此，我们可以看到，PIAS 与 Panda 的性能差距越来越大。在请求量为 10000 时，请求延时的 90% 分位值在使用 Panda 时为 0.6ms，在使用 PIAS 时为 1.7ms，性能提升了 ~65%；请求延时的 99% 分位

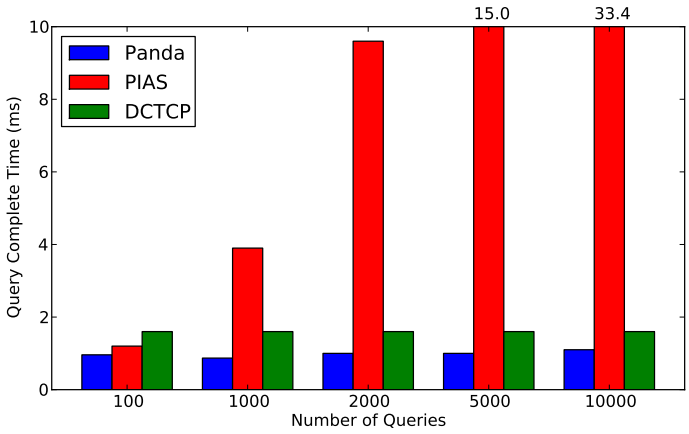


图 5.2　使用 CUBIC 时 Memcached 的请求延时 99% 分位值比较

值在使用 Panda 时为 0.8ms，在使用 PIAS 时为 33.4ms，性能提升了 ~97%。PIAS 性能如此差的原因在于，此时 Memcached 产生的 value 数据包被分配了最低的优先级。由于多优先级队列严格按照优先级的顺序进行调度，而且由于队列长度没有任何限制，导致低优先级队列的排队延时非常严重，所以处于最低优先级队列的数据包经历的延时超过了 30ms。

其次，FLT 数据流大小的增大基本不会影响 Panda 的性能。Panda 使用的补偿函数可以保证其给 FLT 分配的优先级只与数据包大小分布有关，而与数据流中包含的数据包数量无关。因此，我们可以看到，在请求量只有 100 个时，请求延时的 90% 分位值为 0.6ms，99% 分位值为 0.8ms；而当请求量增加到 10000 个时，请求延时的 90% 分位值仍为 0.6ms，99% 分位值约为 0.9ms，与请求量为 100 时的结果基本一致。此外，DCTCP 的性能也与 FLT 数据流的大小无关。其主要原因是，DCTCP 通过设定的阈值将队列的排队长度控制在了一个基本稳定的大小，而与数据流的大小无关。我们从图中可以看出，当只使用 DCTCP 时，请求延时的 90% 分位值为 1.0ms，99% 分位值为 1.6ms。由于在使用 Panda 时，FLT 产生的数据包只需要经历在最高优先级的排队延时，所以 Panda 的性能优于 DCTCP，性能提升了 ~50%。

Panda 与 PIAS 均使用 DCTCP 作为传输层协议时的实验结果如图 5.3 与图 5.4 所示。从这两个图中我们可以得到如下两个结论。

第一，使用 DCTCP 作为传输层协议后，PIAS 的性能提升显著。当请求量为 10000 时，尽管此时 Memcached 仍然被 PIAS 分配了最低优先级，但由于各队列的总排队长度被限制在了一个范围内，数据包需要经历的排队延时大大降低。请求延时的 90% 分位值为 1.1ms，降低了 35%；而 99% 分位值从 33.4ms 降低到了 2.9，降幅超过 91%。

第二，配合使用 DCTCP 后，Panda 的性能仍然最好，且与不使用 DCTCP 时基本一致。其原因是使用 Panda 后，FLT 产生的流量一直处于最高优先级，其数据包仅仅只需要经历最高优先级队列的排队延时。使用 DCTCP 作为传输层协议时，尽管可以有效

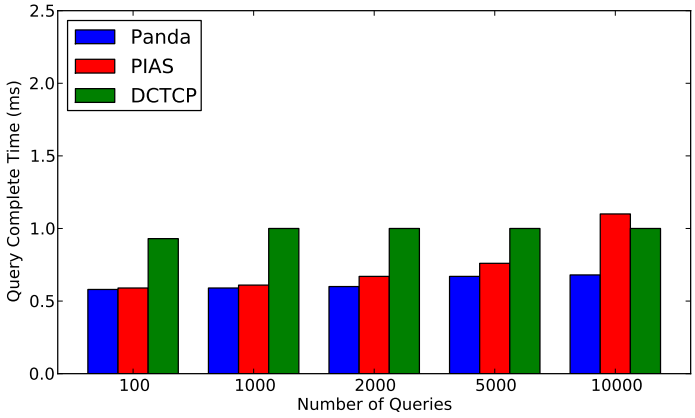


图 5.3 使用 DCTCP 时 Memcached 的请求延时 90% 分位值比较

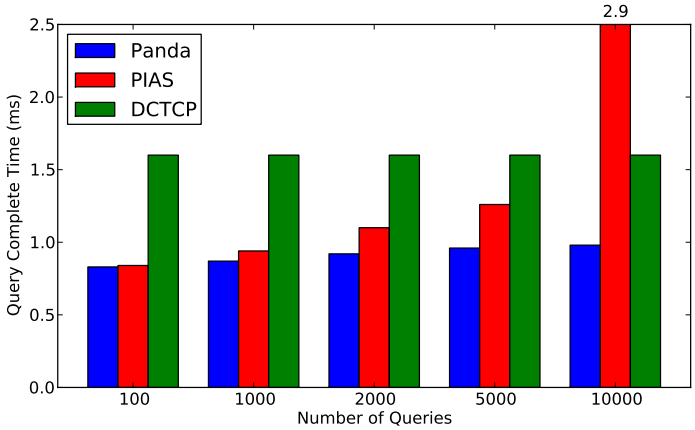


图 5.4 使用 DCTCP 时 Memcached 的请求延时 99% 分位值比较

降低其他较低优先级的队列长度，但无法降低最高优先级的排队延时，所以是否使用 DCTCP 影响并不大。尽管使用 DCTCP 后 PIAS 的性能有了很大提升，但 FLT 产生的数据包仍然会进入低优先级队列排队。所以，Panda 的性能仍优于 PIAS，大约提升了 15~20%。而当 FLT 数据流大小很大时，导致其在使用 PIAS 时落入了最低优先级，对于 99% 分位请求延时，Panda 比 PIAS 优 45~69%。

5.1.1.2 与流量负载的关系

我们评测了 FLT 的请求延时与网络中的流量负载的关系。通过改变网络中数据流的并发度，我么可以调整流量负载的大小。数据流并发度的变化范围是 1~20。由于 FLT 数据流的大小可能与性能相关，我们选择了三种请求量：100,2000 和 10000。随着请求量的增加，PIAS 给 Memcached 数据流分配的优先级越来越低。我们分别评测了在 CUBIC 和 DCTCP 分别作为传输层协议时，Panda 与 PIAS 对于 Memcached 的请求延时的性能，如图 5.5 与 5.6 所示。从实验结果中，我们可以得到如下结论。

第一，随着流量负载的增加，Panda 的性能基本保持稳定，对于 DCTCP 也有这个结论。使用 Panda 时，数据流并发度从 1 上升到 20 时，请求延时的 90% 分位值仅上升了

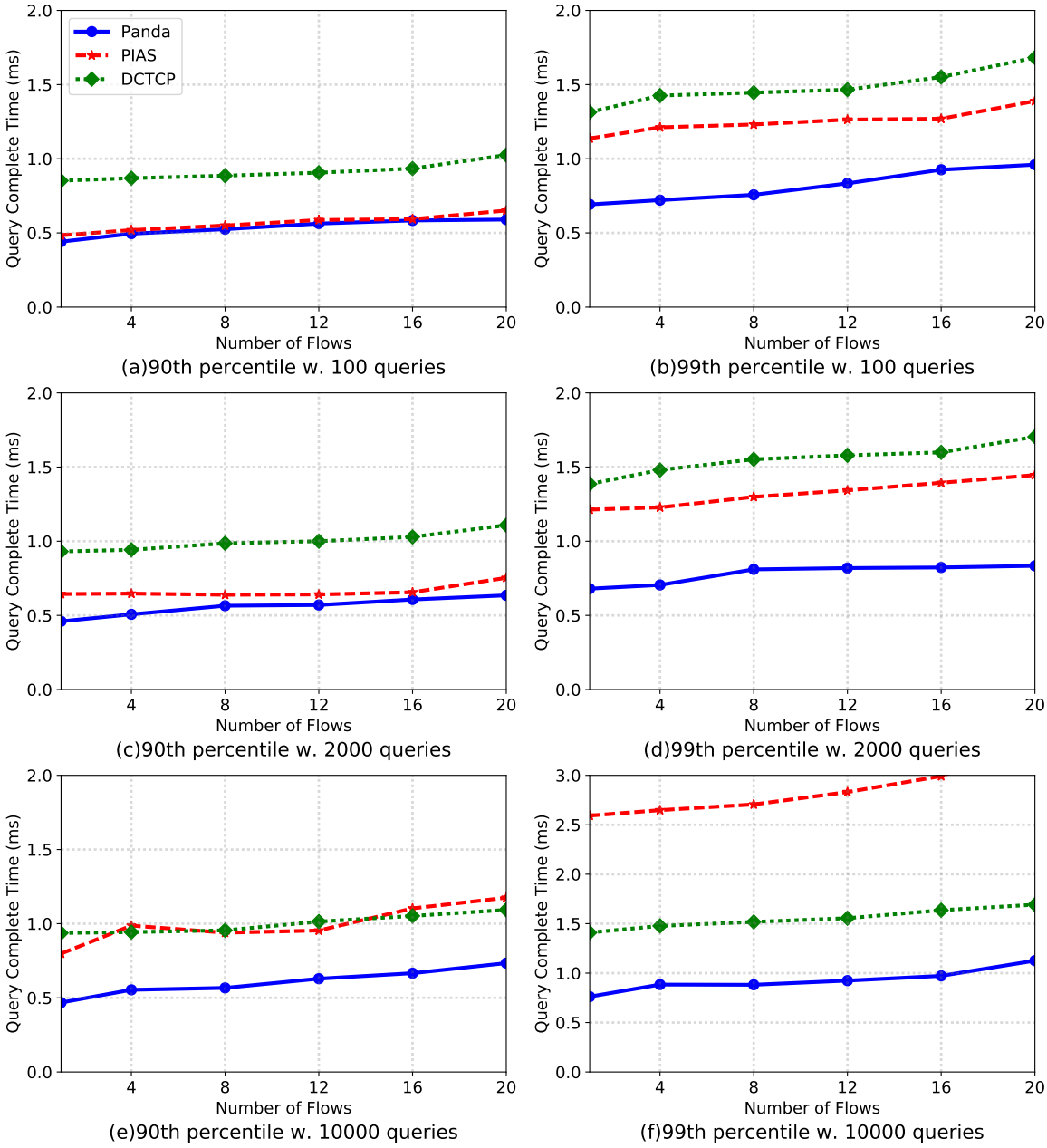


图 5.5　使用 DCTCP 时 Memcached 的请求延时与数据流并发度的关系

不到 0.2ms，99% 分位值也只上升了 0.3ms 左右。原因是虽然数据流并发度增加时，会增大各优先级队列的排队延时，但 Panda 能保证 FLT 一直处于高优先级，所以其性能基本不受影响。由于 DCTCP 能有效地控制队列长度，所以流量负载的增加对其性能影响也较小。

第二，当 DCTCP 作为传输层协议时，PIAS 对流量负载大小不敏感；而当使用 CUBIC 时，负载越大，PIAS 性能越差。配合使用 DCTCP 时，多优先级队列的总队列长度被有效限制在一定范围内，即使流量负载增加，这个范围的波动也很小，所以此时 PIAS 对流量负载不敏感。而没有使用 DCTCP 时，各优先级队列的队列长度没有被约束，所以流量负载越大，队列长度越大。由于 PIAS 会给 Memcached 数据流分配较低的优先级，

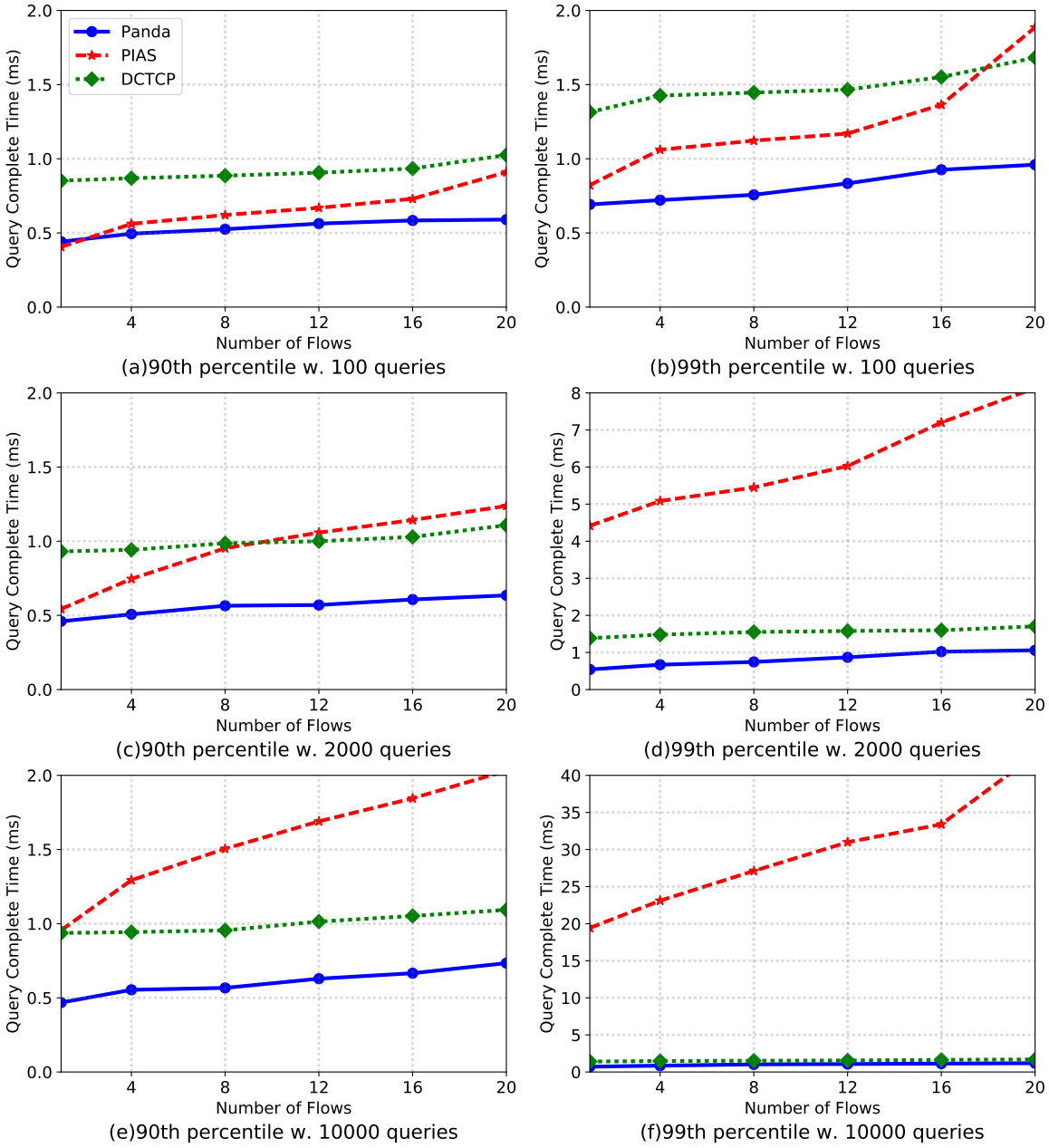


图 5.6 使用 CUBIC 时 Memcached 的请求延时与数据流并发度的关系

导致其产生的数据包在多优先级队列中经历了较大的排队延时，且负载越大，排队延时越长。

5.1.2 10Gbps 网络环境测试

5.1.2.1 与数据流大小的关系

由于在 10Gbps 网络中需要支持 offload 特性才能充分利用带宽，但 ECN 标记大数据包时仍存在一些问题，所以我们没有评测 DCTCP 在 10Gbps 网络环境下的性能。

在 10Gbps 网络中，Panda 与 PIAS 使用 CUBIC 作为传输层协议时的实验结果如图 5.5 与图 5.6 所示。实验中的 FHT 的数据流并发度为 14，此时的流量负载大约为 0.6~0.7。

从这两个图中我们可以得到如下两个结论。

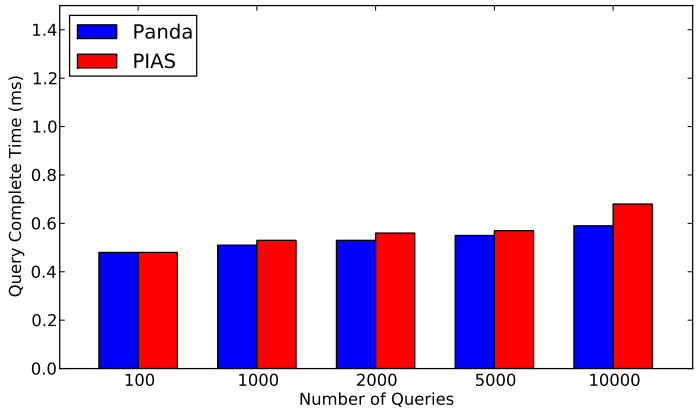


图 5.7　Memcached 的请求延时 90% 分位值比较

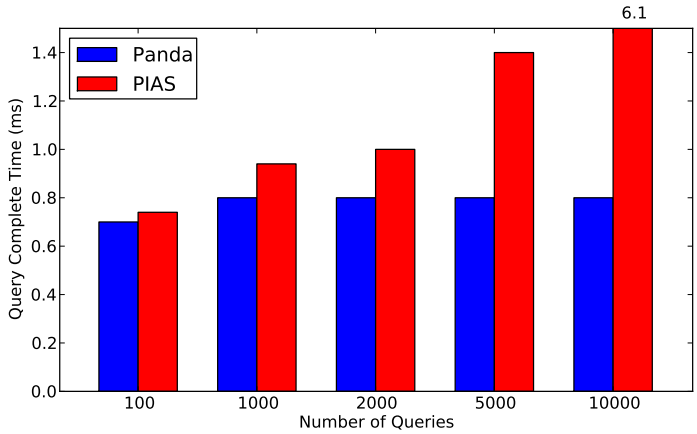


图 5.8　Memcached 的请求延时 99% 分位值比较

首先，10Gbps 网络中，在流量负载居中时（0.6~0.7），网络中的排队延时并不严重。从图 5.5 可以看出，随着 FLT 数据流大小的增加，请求延时 90% 分位值并没有很明显的增加。请求量从 100 增加到 10000 时，在使用 PIAS 的网络中，请求延时 90% 分位值仅有 ~40% 的增加；但长尾延时依然严重，请求延时 99% 分位值增加了 ~88%, 从 0.7ms 增加到了 6.1ms。

其次，Panda 的性能优于 PIAS，且与在 1Gbps 网络环境中的趋势基本一致。在 10Gbps 网络环境中，Panda 的性能并没有随着 FLT 数据流大小的增加而变差。当请求量为 100 时，Panda 与 PIAS 的性能基本一致。但随着请求量的增加，PIAS 的性能逐渐开始恶化，所以 PIAS 与 Panda 的性能差距越来越大。当 PIAS 将一部分 FLT 产生的流量放入最低优先级队列时（请求量为 10000），对于请求延时的 99% 分位值，Panda 比 PIAS 的性能好 ~87%。

5.1.2.2　与流量负载的关系

在 10Gbps 网络环境下，随着网络中流量负载的增加，Panda 与 PIAS 的性能如图 5.9 所示。从图中我们可以得到如下两个结论。

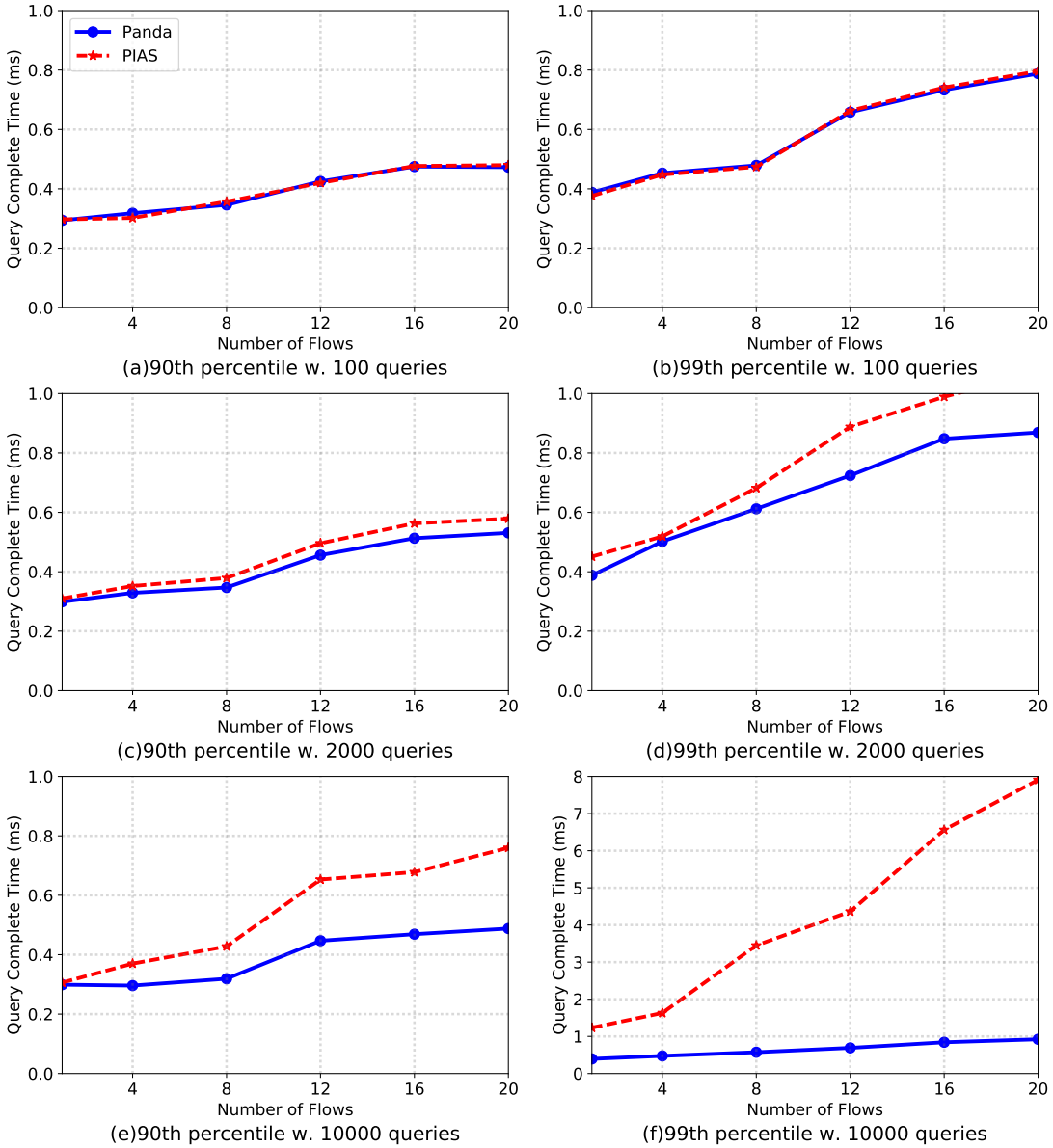


图 5.9　Memcached 的请求延时与数据流并发度的关系

第一，流量负载越大，Panda 相对于 PIAS 的性能优势越大。当请求量为 100 时，由于 Panda 与 PIAS 都会为 Memcached 数据流分配最高优先级，所以两种策略的性能基本一致。当数据流并发度为 1 时，Panda 与 PIAS 的性能差别不大；但随着并发度的增大，PIAS 下的 99% 分位请求延时上升明显。在并发度为 20 时，在 2000 个请求下，比较 99% 分位请求延，Panda 的性能要比 PIAS 好 ~29%；而在 10000 个请求下，Panda 的性能比 PIAS 好 ~87%，此时，即使是 90% 分位请求延时，Panda 的性能也要比 PIAS 好 ~40%。

第二，流量负载的增大会对 Panda 的性能产生影响。数据流的并发度从 1 上升到 20

时，请求延时的 90% 分位值从 0.3ms 上升到了 ~0.5ms，99% 分位值从 0.4ms 上升到了 ~0.8ms。尽管 Panda 能保证 FLT 数据流一直处于最高优先级，但随着流量负载的增加，导致出现在最高优先级队列的数据包也会增加。在没有 offload 特性的网络环境中，数据包大小都是 MTU，少量数据包的增加对排队延时影响不大。但在有 offload 特性的网络中，多个数据包会拼接成大包，进入队列排队，这会对 FLT 产生的数据包的排队延时产生较大影响。

5.2 吞吐密集型应用的平均流完成时间

我们分别测试了在 1Gbps 与 10Gbps 网络环境下的 FHT 平均流完成时间。由于 Panda 并没有区分权值，所以 Panda 与 PIAS 的性能基本一致。

图 5.10 表示的是在 1Gbps 网络下，以 DCTCP 作为传输层协议时 Panda 与 PIAS 的性能。从图中可以看出，与 DCTCP 比较，Panda 与 PIAS 能有效降低 FHT 的平均流完成时间。当数据流并发度为 1 时，三种策略的性能基本一致，而随着流量负载的增加大，Panda 与 PIAS 的性能优势越来越明显。当并发度为 20 时，与 DCTCP 比较，对于小流，，Panda 与 PIAS 可以将平均流完成时间降低 ~21%；而对于中流，平均流完成时间降低了 ~20%。

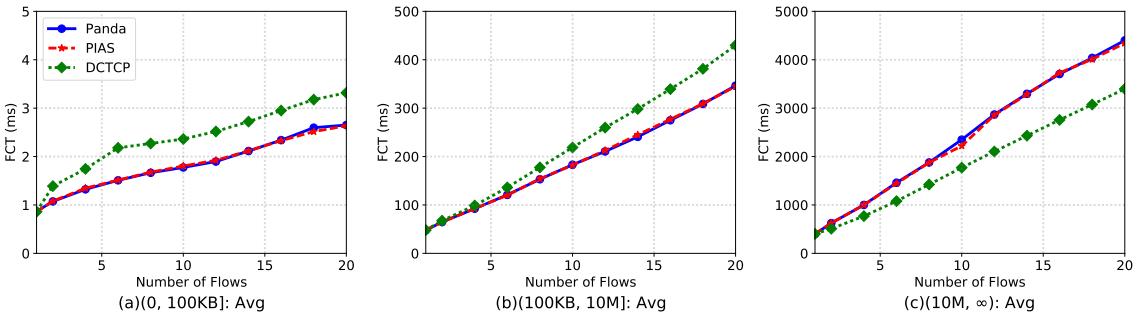


图 5.10　1Gbps 环境下的平均流完成时间

图 5.11 表示的是在 10Gbps 网络下，以 CUBIC 作为传输层时 Panda 与 PIAS 的性能，两者性能基本一致。

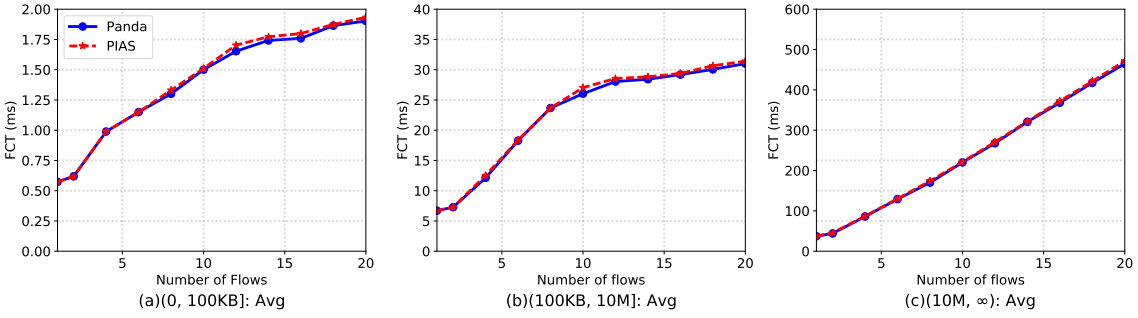


图 5.11　10Gbps 环境下的平均流完成时间

5.3 小结

本章主要描述了 Panda 在 1Gbps 网络环境与 10Gbps 网络环境下，对 FLT 的延时保证性能和对 FHT 的平均流完成时间优化性能。我们将 Panda 与 DCTCP 和 PIAS 都进行了对比。实验结果表明，与 DCTCP 和 PIAS 比较，Panda 能有效降低 FLT 所经历的终端排队延时，且 FLT 数据流大小越大，性能优势越明显。与此同时，在优化平均流完成时间方面，PIAS 与 Panda 有着基本一致的性能。

第六章　结束语

6.1　工作总结

本论文主要讨论了如何在数据中心内，在不需要提前预知数据流信息的前提下，如数据流大小、应用类型和请求时间限制等，同时保证延时敏感型应用产生的流量的低延时和优化吞吐密集型应用产生的数据流的平均流完成时间。本论文提出的 Panda 方案通过分析数据中心的流量特征，主要是不同应用产生的数据流中数据包大小的分布特征，找到了能区分延时敏感型应用与吞吐密集型应用的阈值。该阈值能保证吞吐密集型应用产生的流量中大部分都是大包，而小包却在延时敏感型应用中占主要部分。Panda 通过数据流已经发送的数据包为其分配优先级，并使用多优先级队列对数据流进行调度。当数据包大小大于阈值时，Panda 对该数据流的优先级进行惩罚，而小于阈值时，则对其优先级进行补偿。通过这种方案，Panda 可以在不需要提前预知数据流信息的前提下同时保证两种应用的服务质量。

我们在 Linux 平台上实现了一个 Panda 原型，并评估了 Panda 的性能。实验结果显示，Panda 能很好地实现我们的设计目标。

6.2　下一步研究计划

Panda 还有三个不够完善的方面。

第一个方面是如何分别获取延时敏感型应用和吞吐密集型应用的数据包大小分布。在未使用 offload 特性的网络环境下，数据包大小可以在交换机处得到。数据中心服务商可以在交换机处获取数据流的数据包大小信息。但在使用了 offload 特性的网络环境中，数据包在终端的大小与在到达交换机时的大小不一致。所以，只有在终端才能得到数据包大小的信息。因此，每台终端都需要安装一个获取数据包大小的监控软件。由于 Panda 也同样需要安装在每台终端，所以 Panda 可以同时实现数据包大小信息的抓取，但这部分工作 Panda 还没实现。获取到数据流的数据包大小分布后，如何确定该数据流的属性是接下来需要考虑的问题。我们需要将该数据流标记为延时敏感或是吞吐密集，但如何标记仍是一个问题。虽然本论文假设已经知道了 FLT 和 FHT 的数据包大小分布，但这个分类过程实际上并不简单。我们可以要求应用提供其产生的数据流的属性，但这会存在两个潜在问题：1）用户提供的信息不一定准确；2）用户可能并不知道其应用产生的数据流的真实属性。因此，我们仍需要其他的方式进行区分。

第二个方面是如何在有 offload 特性的网络环境中配合使用 ECN 标记降低终端的排队延时。由于数据包在离开终端前很有可能会是拼接了多个 MSS 的大数据包，其大小甚至可以达到 64KB，所以 [5] 中推荐的 ECN 标记阈值不再适用，需要找到新的 ECN 标

记阈值来优化网络性能。其次是如何解决 ECN 标记比例过大的问题。当一个大数据包在终端被打上 ECN 标记后，网卡在将这个大数据包分成大小为 MTU 的小数据包时，每个数据包都会被打上 ECN 标记，导致被标记了 ECN 的数据包的比例比实际超过阈值的数据包比例要大。由于 ECN 打标比例过大，导致拥塞窗口大小减小过快，可能会影响网络带宽的充分利用。

第三个方面是如何选取优化 FHT 平均流完成时间的权值。由于时间有限，本论文的工作并没有为如何寻找最优权值建立最优化模型，但根据数据流的数据包大小分布，应该能确定最优的权值，让大流更快地落入低优先级，从而实现进一步优化平均流完成时间的目标。

参考文献

[1] T. A. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *ACM SIGCOMM Workshop: Research on Enterprise Networking*, January 2009.

[2] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 267–280, ACM, 2010.

[3] G. Judd, "Attaining the promise and avoiding the pitfalls of TCP in the datacenter," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 145–157, 2015.

[4] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," *ACM SIGCOMM 2015*, vol. 45, no. 5, pp. 123–137, 2015.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," *ACM SIGCOMM 2010*, vol. 40, no. 4, pp. 63–74, 2010.

[6] C. Hong, M. Caesar, and P. B. Godfrey, "Finishing flows quickly with preemptive scheduling," *ACM SIGCOMM 2012*, vol. 42, no. 4, pp. 127–138, 2012.

[7] B. Vamanan, J. Hasan, and T. N. Vijaykumar, "Deadline-aware datacenter TCP (D2TCP)," *ACM SIGCOMM 2012*, vol. 42, no. 4, pp. 115–126, 2012.

[8] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: meeting deadlines in datacenter networks," *ACM SIGCOMM 2011*, vol. 41, no. 4, pp. 50–61, 2011.

[9] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 1–14, 2015.

[10] L. Chen, K. Chen, W. Bai, and M. Alizadeh, "Scheduling mix-flows in commodity datacenters with Karuna," in *ACM SIGCOMM 2016*, pp. 174–187, 2016.

[11] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. Mckeown, B. Prabhakar, and S. Shenker, "pFabric: minimal near-optimal datacenter transport," *ACM SIGCOMM 2013*, vol. 43, no. 4, pp. 435–446, 2013.

[12] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 455–468, 2015.

[13] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar, "Friends, not foes: synthesizing existing transport strategies for data center networks," *ACM SIGCOMM 2014*, vol. 44, no. 4, pp. 491–502, 2014.

[14] I. A. Rai, G. Urvoykeller, M. K. Vernon, and E. Biersack, "Performance analysis of LAS-based scheduling disciplines in a packet switched network," *Measurement and Modeling of Computer Systems*, vol. 32, no. 1, pp. 106–117, 2004.

[15] R. Fielding, "RFC 2068: Hypertext Transfer Protocol-HTTP/1.1," *ftp://ftp.internic.net/rfc/rfc2068.txt*, vol. 7, no. 4, pp. 237–264, 1997.

[16] "Microsoft SQL Server." http://www.microsoft.com/en-us/server-cloud/products/sql-server/.

[17] "Apache Storm." http://storm.apache.org/.

[18] R. Mittal, N. Dukkipati, E. Blem, H. M. G. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, "TIMELY: RTT-based congestion control for the datacenter," *ACM SIGCOMM 2015*, vol. 45, no. 4, pp. 537–550, 2015.

[19] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: a centralized "zero-queue" datacenter network," *ACM SIGCOMM 2014*, vol. 44, no. 4, pp. 307–318, 2014.

[20] "VMware." http://www.vmware.com/.

[21] "Xen." http://xenproject.org/.

[22] "KVM." http://www.linux-kvm.org/.

[23] "Linux container." https://linuxcontainers.org/.

[24] "Memcached." https://memcached.org/.

[25] "iPerf." https://iperf.fr/.

[26] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.

[27] K. He, E. Rozner, K. B. Agarwal, W. Felter, J. B. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," *ACM SIGCOMM 2015*, vol. 45, no. 4, pp. 465–478, 2015.

[28] R. Kapoor, A. C. Snoeren, G. M. Voelker, and G. Porter, "Bullet trains: a study of nic burst behavior at microsecond timescales," in *ACM Conference on Emerging NETWORKING Experiments and Technologies*, pp. 133–138, 2013.

[29] "Linux tc pfifo." https://linux.die.net/man/8/tc-pfifo.

[30] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 4, pp. 397–413, 1993.

[31] K. Nichols and V. Jacobson, "Controlling queue delay," *ACM Queue*, vol. 10, no. 5, p. 20, 2012.

[32] M. Miao, P. Cheng, F. Ren, and R. Shu, "Slowing little quickens more: Improving DCTCP for massive concurrent flows," *IEEE ICPP*, Sept 2015.

[33] W. Bai, K. Chen, H. Wu, W. Lan, and Y. Zhao, "PAC: Taming TCP incast congestion using proactive ack control," *IEEE ICNP*, Oct 2014.

[34] "Linux tc fq codel." http://man7.org/linux/man-pages/man8/tc-fq_codel.8.html.

[35] "Aliyun." https://www.aliyun.com/.

[36] "Amazon Website Services." https://aws.amazon.com/.

[37] "Hadoop." http://hadoop.apache.org/.

[38] "WeChat." https://weixin.qq.com/.

[39] "Exponentially Weighted Moving Average." https://en.wikipedia.org/wiki/Moving_average.

[40] W.-c. Feng, K. G. Shin, D. D. Kandlur, and D. Saha, "The blue active queue management algorithms," *IEEE/ACM Transactions on Networking*, vol. 10, pp. 513–528, Aug. 2002.

[41] "CoDel pseudo." http://queue.acm.org/appendices/codel.html.

[42] K. Ramakrishnan, S. Floyd, and D. Black, "The addition of explicit congestion notification (ECN) to IP," 2001.

[43] A. Munir, I. A. Qazi, Z. A. Uzmi, A. Mushtaq, S. N. Ismail, M. S. Iqbal, and B. Khan, "Minimizing flow completion times in data centers," in *INFOCOM*, 2013.

[44] J. Nash, "Equilibrium points in n-person games," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 36, no. 1, pp. 48–49, 1950.

[45] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, “Less is more: trading a little bandwidth for ultra-low latency in the data center,” in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI 2012)*, USENIX Association, 2012.

[46] “Openstack.” https://www.swiftstack.com/product/openstack-swift.

[47] B. Kalyanasundaram and K. Pruhs, “Minimizing flow time nonclairvoyantly,” *Journal of the ACM*, vol. 50, no. 4, pp. 551–567, 2003.

[48] “Linux netfilter.” http://www.netfilter.org.

[49] W. Bai, L. Chen, K. Chen, and H. Wu, “Enabling ecn in multi-service multi-queue data centers,” in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI 16)*, pp. 537–549, 2016.

[50] “Linux kernel.” http://www.kernel.org.

[51] “Linux epoll.” http://man7.org/linux/man-pages/man7/epoll.7.html.

附录 A　CoDel 伪代码

CoDel 伪代码中使用到的数据类型如表 1 所示。

表 1. CoDel 伪代码中的数据类型说明

数据类型	说明
time_t	整型
flag_t	布尔型
packet_t	指向数据包描述符的指针
queue_t	队列基类，包含 enqueue() 和 dequeue()
codel_queue_t	CoDel 队列类，继承 queue_t

CoDel 伪代码中使用到的参数及其含义说明如表 2 所示。

表 2. CoDel 伪代码中的参数说明

数据类型	参数	说明
time_t	target	排队延时阈值，常量，默认值为 5ms
time_t	interval	持续时间阈值，常量，默认值为 100ms
time_t	first_above_time	排队延时开始大于 target 的时间
time_t	drop_next	丢弃下一个数据包的时间
flag_t	dropping	为 1 时表示进入丢包状态
uint	count	进入丢包状态后的丢包个数
uint	maxpacket	数据包大小的最大值，常量，设为 MTU

当数据包进入队列时，调用 enqueue() 函数。CoDel 在此时给数据包打上时间戳，用于记录数据包在队列的排队延时。

```
void codel_queue_t::enqueue(packet_t* pkt)
{
    pkt->timestamp() = clock();
    queue_t::enqueue(pkt);
}
```

由于经过该队列的数据流的并发度和发送端的属性未知，CoDel 需要设计成一个闭环回路反馈系统，它逐渐增加丢包频率直到数据包的排队延时被控制到小于 *target*。*control_law()* 用于控制丢包频率。由于该函数的返回值与 *count* 的平方根成反比，因此当进入丢包状态后，丢包数量越多，丢包频率越高，以便于排队延时能更快地降低到小于 *target*。

```
time_t codel_queue_t::control_law(time_t t)
{
    return t + interval / sqrt(count);
}
```

在数据包离开队列前，需要判断该数据包是否需要丢弃。*dodequeue()* 用于判断数据包的排队延时大于 *target* 的持续时间是否长于 *interval*。如果不足 *interval*，说明该数据包不需要丢弃；如果大于 *interval*，则还需要进一步判断。

```
typedef struct {
    packet_t* p;
    flag_t ok_to_drop;
} dodequeue_result;

dodequeue_result codel_queue_t::dodequeue(time_t now)
{
    dodequeue_result r = { 0, queue::dequeue() };
    if (r.p == NULL) {
        first_above_time = 0;
    } else {
        time_t sojourn_time = now - r.p->tstamp;
        if (sojourn_time < target || bytes() < maxpacket) {
            first_above_time = 0;
        } else {
            if (first_above_time == 0) {
                first_above_time = now + interval;
            } else if (now >= first_above_time) {
                r.ok_to_drop = 1;
            }
        }
    }
    return r;
}
```

dequeue() 是 CoDel 工作的主要函数。如果数据包的排队延时大于 *target* 的持续时间小于 *interval*，说明队列还没有进入丢包状态，数据包不需要被丢弃。

```
packet_t* codel_queue_t::deque()
```

```
{
    time_t now = clock();
    dodeque_result r = dodeque();
    if (r.p == NULL) {
        // leaving dropping state
        dropping = 0;
        return r.p;
    }
    if (dropping) {
        if (! r.ok_to_drop) {
            // leave dropping state
            dropping = 0;
        } else if (now >= drop_next) {
```

如果数据包的排队延时大于 *target* 的持续时间大于 *interval*，且到了丢弃下一个数据包的时间，CoDel 开始丢包。

```
            while (now >= drop_next && dropping) {
                drop(r.p);
                ++count;
                r = dodeque();
                if (! r.ok_to_drop)
                    // leave dropping state
                    dropping = 0;
                else
                    // schedule the next drop.
                    drop_next = control_law(drop_next);
            }
        }
```

即使现在并没有处于丢包状态，在某些情况下，CoDel 会直接进入丢包状态，并丢弃数据包。

```
    } else if (r.ok_to_drop &&
               ((now - drop_next < interval) ||
                (now - first_above_time >= interval))) {
        drop(r.p);
        r = dodeque();
```

```
        dropping = 1;

        if (now - drop_next < interval)
            count = count>2? count-2 : 1;
        else
            count = 1;
            drop_next = control_law(now);
        }
    return (r.p);
}
```

附录 B　最优化问题

最小化如下目标函数：

$$\begin{aligned} \underset{\{\alpha_i\}}{min} \quad & T = \sum_{j=1}^{K} \theta_j \sum_{i=1}^{j-1} U_i + \sum_{j=1}^{K} \frac{1}{2} \theta_j U_j \\ \text{suject to} \quad & \alpha_0 = 0, \alpha_K = x_{max} \\ & \alpha_{j-1} < \alpha_j, j = 1, 2, \cdots, K \end{aligned}$$

其中，$U_i = (\alpha_i - \alpha_{i-1}) T_i$，$i \in [1, K]$。

B.1　目标函数

为了方便求解，我们将原目标函数近似成

$$\begin{aligned} \underset{\{\alpha_i\}}{min} \quad T &\approx \sum_{j=1}^{K} \theta_j \sum_{i=1}^{j-1} U_i + \sum_{j=1}^{K} \theta_j U_j \\ &= \sum_{j=1}^{K} \theta_j (\sum_{i=1}^{j} U_i) \\ &= \sum_{j=1}^{K} (U_i \sum_{i=1}^{j} \theta_j) \end{aligned}$$

我们使用 θs 等价替代 αs。假设网络的流量负载密度为 ρ，排队模型为 M/M/1，那么我们可以解得 $U_l = \frac{\theta_l \rho}{1 - \rho F(\alpha_{l-1})}$。由于

$$\sum_{m=1}^{l} \theta_m = \sum_{m=l}^{K} F(\alpha_m) - F(\alpha_{m-1})$$

我们能把目标函数转化为

$$T = \sum_{l=1}^{K} T_l (1 - F(\alpha_{l-1}))$$

由于 $\sum_{l=1}^{K} \theta_l = 1$，我们最终将最优化问题转换成

$$\underset{\{\alpha_i\}}{max} \quad T = \sum_{l=1}^{K-1} \frac{\theta_l}{1 - \rho(\sum_{m=1}^{l-1 \theta_m})} + \frac{1 - \sum_{m=1}^{K-1} \theta_m}{1 - \rho \sum_{m=1}^{K-1} \theta_m}$$

这是一个 Sum-of-linear-Ratios(SoLR) 问题，是一个典型的非整数线性规划问题。上式说明，其上界仅与 θs 有关（优先级阈值的相对值），而与数据流大小分布函数无关。因此，我们可以先求出 θs，然后根据 $F(x)$ 得到优先级阈值的具体值。

B.2 近似算法

SoLR 问题是一类 NP 难的问题。解决这类问题的难点通常在于目标函数缺乏有用的性质。对于上述问题，我们能找到一些合适的性质求解该最优化问题的闭环解析解（closed-form analytical solution）。

由于流量负载 $\rho \leq 1$，所以我们有

$$\rho(\sum_{m=1}^{l-1}\theta_m) \leq (\sum_{m=1}^{l-1}\theta_m)$$

此外，由于

$$\theta_l + \sum_{m=1}^{l-1}\theta_m = \sum_{m=1}^{l}\theta_m \leq 1$$

所以有

$$\theta_l \leq \sum_{m=l}^{K}\theta_m = 1 - \sum_{m=l}^{l-1}\theta_m \leq 1 - \rho(\sum_{m=1}^{l-1}\theta_m)$$

由于 $\theta_l/(1-\rho(\sum_{m=1}^{l-1}\theta_m)) \leq 1$，为了求出最大值，我们需要保证式子中分子与分母尽可能接近，才能使比值越来越接近于 1。

先考虑前两部分 θ_1 与 θ_2，通过使分子与分母相等，我们可以得到

$$\theta_2 = 1 - \rho\theta_1$$

我们也能通过迭代的方式得到第三部分 θ_3 的值：

$$\theta_2 = 1 - \rho(\theta_1 + \theta_2) = 1 - \rho(1 - (1-\rho)\theta_1)$$

通过迭代公式

$$\theta_l = 1 - \rho(\sum_{m=1}^{l-1}\theta_1)$$

我们可以将所有的 θ_l 表示成关于 θ_1 的表达式。此外，由于 $\sum_{l=1}^{K}\theta_l = 1$，所以我们能解出所有的 θs。再根据已知的数据流大小分布函数 $F(x)$，便能求解出所有的优先级阈值。

致　谢

衷心感谢我的导师陈明宇研究员和刘珂助理研究员对本人的悉心指导。陈老师治学异常严谨。每次论文讨论班，陈老师都会带着我们分析论文的贡献点，组织结构和实验组成。陈老师总是教导我们，“虽然现在灌水论文很多，但我希望你们产出的每篇论文都能对学术圈有实际的贡献，而不是为了发论文而发论文”。陈老师知识渊博，每次与他讨论问题他总能直击要害。他不仅对我的研究工作提供了很多宝贵的建议，也教会了我许多做人的道理，使我受益颇多。刘老师与我亦师亦友，感谢刘老师这两年来对我学术上的指导和生活上的关心。刘老师在学术上为我提供了莫大的支持，我有任何学术方面的问题都可以直接与他讨论，甚至都不需要预约时间。每次修改论文时，刘老师都会要求我坐在他旁边，一句一句地帮我把那些像是用谷歌翻译过来的句子改得具有学术范。由于年龄相仿，有段时间我们基本上每天中午都一起吃“京味张”（还有陈威屹），聊聊最近发生的八卦和趣事，让我感觉很亲切。

感谢张文力老师对我学习生活的支持与鼓励。在攻关院先导海云项目时，张老师为我们提供了从软件到硬件全方位的支持，陪我们一起加班加点调代码，为了项目能顺利通过验收付出大量的时间和精力。在生活中，张老师多次与我促膝长谈，聊聊人生，谈谈理想，同时也为我的研究生生活提供了方方面面的帮助。

感谢吴洁师姐对我生活和学术上的照顾。吴洁师姐与我本科都毕业于华中科技大学计算机系，在我研究生期间，一直像照顾弟弟一样照顾我，中午带我吃饭，买了好吃的零食也会与我分享。学术上，师姐经常与我交流 OVS 与 SDN 领域的知识，使我增长了不少见识。我们俩也会讨论网络协议栈相关的问题，使我受益匪浅。

感谢组里的师兄和师弟，将我的研究生生活变得丰富多彩。李龙师兄对中国传统文化有其独到的见解，每次与师兄聊天都仿佛打开的一片新天地，也让自己体会到了“书到用时方恨少”的愧疚。沈逸凡师弟则是天真无敌，即便是天立马要塌下来了，也得让他先吃口饭；感谢师弟长期赞助的水果与干粮。陈威屹同学“天上知道一半，地上全知道”，时时刻刻为我们带来着欢乐。非常感谢网络组的所有老师、同学及员工，三年来大家对我一直非常关照，让我能在这么轻松、自在、快乐的氛围中学习生活。我非常享受这三年来和大家一起度过的岁月，是大家，让这段时光变得如此地难忘也不平凡。

我要感谢我的妻子，一直陪伴着我，与我一同为了更美好的明天奋斗着。虽然我们俩都身在异乡，但依然能时时刻刻感受到家的温暖。感谢她对我不求回报地默默付出，使我能心无旁骛地在实验室跑实验、做项目、赶论文。我还要感谢我的家人对我的理解，你们的支持是我前进的最大动力。

最后，我要衷心的感谢各位专家老师在百忙之中抽出宝贵时间来审阅我的论文参加

我的答辩，谢谢您们！

作者简介

姓名：王壮　　性别：男　　出生日期：1993.06.10　　籍贯：湖北

2014.09 – 2017.07　　中国科学院计算技术研究所硕士生
2010.09 – 2014.06　　华中科技大学计算机科学与技术专业本科生

【攻读硕士学位期间发表的论文】

[1] Zhuang Wang, Ke Liu, Long Li, Weiyi Chen, Mingyu Chen, Lixin zhang, “A Novel Approach for All-to-All Routing in All-optical Hypersquare Torus Network,” in Proc. of ACM International Conference on Computing Frontiers (CF), 2016.

[2] Ke Liu, Zhuang Wang, Jack Y. B. Lee, Mingyu Chen, Lixin Zhang, “Adaptive Rate Control over Mobile Data Networks with Heuristic Rate Compensations,” in Proc. of IEEE/ACM International Symposium on Quality of Service (IWQoS), 2016.

[3] Zhuang Wang, Ke Liu, Yifan Shen, Jack Y. B. Lee, Mingyu Chen, Lixin Zhang, “Intra-host Rate Control with Centralized Approach,” in Proc. of IEEE Cluster 2016, short paper.

【攻读硕士学位期间参与的科研项目】

[1] 中国科学院战略性先导科技专项（XDA06010401）

[2] 华为 A 类高通量服务器项目（YBCB2011030）

【攻读硕士学位期间的获奖情况】

[1] 2016 年被评为中国科学院大学“三好学生”

[2] 2016 年获得国家奖学金