## NDP协议设计详细说明文档

## 一、NDP协议概述

## （一）简介

通过冗余的Clos拓扑结构，现代数据中心网络可以提供非常高的吞吐和低交换延迟，但是传输协议却很少能提供与之相匹配的性能。

全新的NDP协议，和现有的其他传输协议相比，在各种脚本中的短传输、高流量模型以及incast模式下，实现了近乎最优的成功次数。NDP协议的切换缓冲区是非常浅的，当他们填充这些切换时，把修整的数据包给头部位置并且放置优于之前的数据包，接受者拥有一个来自所有发送者即时需求的完整视角，并且这一特点，是该协议在RTT周期中，应对来自不同发送者的优先传输处理请求以及其他请求的基础。实验在各种复杂环境下，证明了它低延迟和高吞吐的优秀表现。

## （二）主要表现

1.比DCTCP或DCQCN有更好的短流表现。

2. 只有8个数据报的交换队列却在高负载情况下比95%的大型网络表现好。

3. 在Incast案例下，拥有近乎完美的低延迟和公平。

4.通往不同主机的流之间的干扰最小。

5. 在incast期间，有效的对于掉队流量的优先考虑。

## 二、NDP协议设计

## （一）协议模型

1.启动

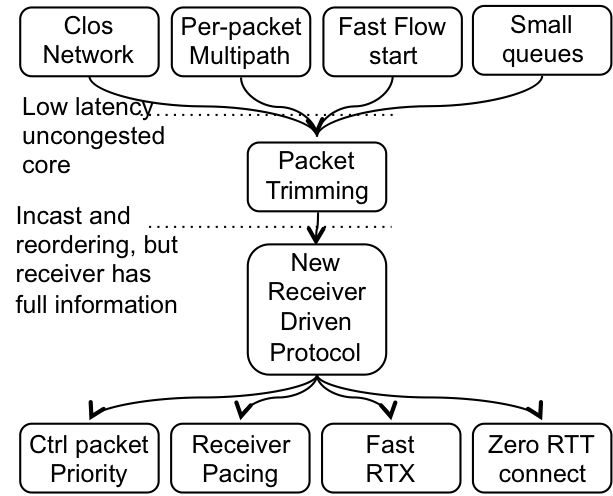
为了实现最小的延迟和吞吐，最初以线路速率启动发送，并且没有连接握手建立，省去了传统的握手时间。

2.离散化

通过将数据报离散化，整合了ecmp，使一段数据能够离散化经过所有可用路径到达接收方。

3.数据修剪及pull方式

建立8个数据包的数据交换队列，以及单独的优先头发送队列。当数据交换队列溢出时，溢出的数据包头被修剪下来，通过头发送队列抵达接收者，使接收者尽早知道发生了什么，在数据交换队列有空闲时，控制被修剪的数据快速重传。之后建立pull队列，以pull的方式控制接下来数据发送速率。pull队列建立之后,由于传输速率受到控制，修剪也不再发生。



NDP协议模型架构图

## （二）测试平台

1.htsim模拟器

2.使用DPDK的LINUX终端系统

3.使用DPDK的经典LINUX交换机

4.使用NETFPGA SUME的硬件交换机

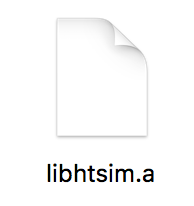
5.P4 reference 交换机

## 三、实验流程

由于条件受限，以下实验是基于htsim模拟器实现的。

## （一）模拟器建立

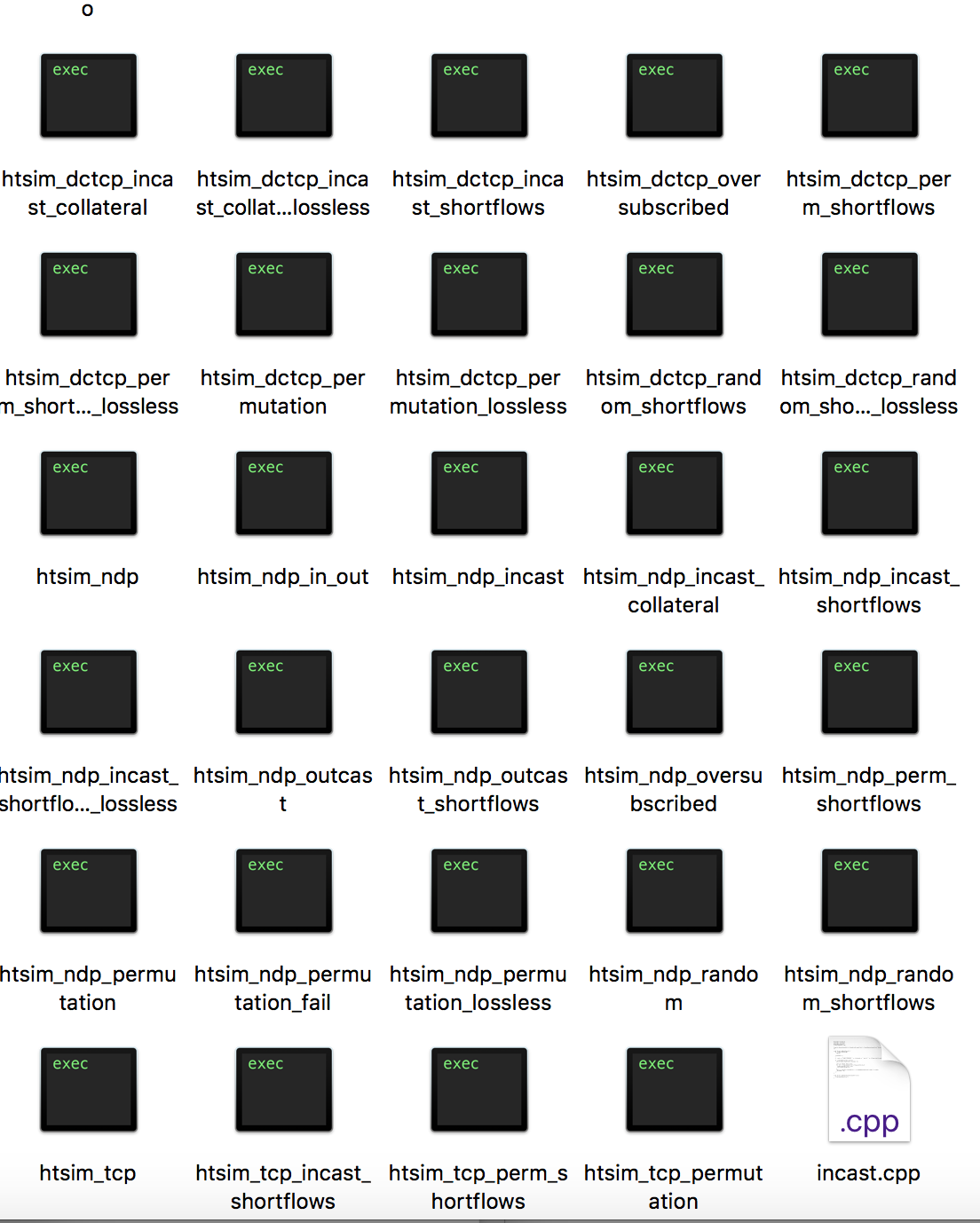
1.在sim文件夹中run make，编译了tcp,ndp,htsim和网络源文件。输出得到libhtsim.a；



2.run make parse\_output得到输出程序（parse\_output）；



3.进入datacenter文件夹，run make。编译了所有的htsim支持的数据中心拓扑，生成了一些执行文件（叫htsim…）用来运行不同实验的。可执行文件Htsim\_X来自于main\_X源文件，用来建立和运行实验。



## （二）简单的协议吞吐率测试

**1.NDP协议吞吐率表现**

./htsim\_ndp\_permutation -strat perm -nodes 16 -conns 16 -cwnd 30

生成一个胖树拓扑（包含16个服务器，k=4），所有链路以10Gbps的速率运行。实验会运行一个包含16个连接的组合流量模型，每个服务器发送和接受一个长的运行的NDP连接，NDP发送者使用的原始窗口设定为30.最后，数据报使用组合的策略通过可用的路线传播出去：每个源将发送数据包以一个随机组合的路线，用轮询的方式。在所有路线都被访问后，排列重新产生。这确保了在网络核心的任何一个端口没有长时间存在的拥塞。

默认情况下，所有的实验都会输出结果在一个名叫logout.dat的文件中。为了分析这些结果，run下面的命令：

../parse\_output logout.dat -ndp -show

输出如下：

9857.16 Mbps val 354 name ndp\_sink\_15\_13(0)

9856.08 Mbps val 350 name ndp\_sink\_14\_1(0)

9855.36 Mbps val 346 name ndp\_sink\_13\_9(0)

9856.80 Mbps val 342 name ndp\_sink\_12\_14(0)

9855.36 Mbps val 338 name ndp\_sink\_11\_3(0)

9855.00 Mbps val 334 name ndp\_sink\_10\_7(0)

9855.72 Mbps val 330 name ndp\_sink\_9\_15(0)

9856.80 Mbps val 326 name ndp\_sink\_8\_10(0)

9855.72 Mbps val 322 name ndp\_sink\_7\_2(0)

9857.52 Mbps val 318 name ndp\_sink\_6\_5(0)

9859.32 Mbps val 314 name ndp\_sink\_5\_4(0)

9855.72 Mbps val 310 name ndp\_sink\_4\_8(0)

9855.36 Mbps val 306 name ndp\_sink\_3\_6(0)

9855.72 Mbps val 302 name ndp\_sink\_2\_12(0)

9859.32 Mbps val 298 name ndp\_sink\_1\_0(0)

9856.08 Mbps val 294 name ndp\_sink\_0\_11(0)

Mean of lower 10pc (1 entries) is 1231875000.000000 total mean 1232055000.000000 mean2 0.000000

每一行给出了每条连接的吞吐量，后面接着是这条连接的细节包括它在trace文件夹的id和它的名字。这个名字也包含这条连接的发送方接收方。最后一行给了每秒比特数的平均值（较低的百分之十的流），以及总的平均值（1100Mbps）。

可以操作模拟器通过提高拓扑大小（最高达8192个节点），改变初始窗口，增加连接数量（从利用率的观点看，连接数和服务器数量相等时是最差的情况）。也可以操作路由策略（模拟的ecmp）

**2.tcp协议吞吐率表现**

77.41 Mbps val 2692 name ndp\_sink\_127\_0(0)

45.16 Mbps val 2689 name ndp\_sink\_126\_0(0)

74.19 Mbps val 2686 name ndp\_sink\_125\_0(0)

77.41 Mbps val 2683 name ndp\_sink\_124\_0(0)

77.77 Mbps val 2680 name ndp\_sink\_123\_0(0)

77.77 Mbps val 2677 name ndp\_sink\_122\_0(0)

77.77 Mbps val 2674 name ndp\_sink\_121\_0(0)

77.77 Mbps val 2671 name ndp\_sink\_120\_0(0)

78.13 Mbps val 2668 name ndp\_sink\_119\_0(0)

77.77 Mbps val 2665 name ndp\_sink\_118\_0(0)

77.77 Mbps val 2662 name ndp\_sink\_117\_0(0)

78.13 Mbps val 2659 name ndp\_sink\_116\_0(0)

77.77 Mbps val 2656 name ndp\_sink\_115\_0(0)

77.77 Mbps val 2653 name ndp\_sink\_114\_0(0)

77.77 Mbps val 2650 name ndp\_sink\_113\_0(0)

77.41 Mbps val 2647 name ndp\_sink\_112\_0(0)

77.77 Mbps val 2644 name ndp\_sink\_111\_0(0)

78.13 Mbps val 2641 name ndp\_sink\_110\_0(0)

78.13 Mbps val 2638 name ndp\_sink\_109\_0(0)

78.13 Mbps val 2635 name ndp\_sink\_108\_0(0)

78.49 Mbps val 2632 name ndp\_sink\_107\_0(0)

78.13 Mbps val 2629 name ndp\_sink\_106\_0(0)

78.13 Mbps val 2626 name ndp\_sink\_105\_0(0)

78.49 Mbps val 2623 name ndp\_sink\_104\_0(0)

78.13 Mbps val 2620 name ndp\_sink\_103\_0(0)

78.13 Mbps val 2617 name ndp\_sink\_102\_0(0)

78.13 Mbps val 2614 name ndp\_sink\_101\_0(0)

78.13 Mbps val 2611 name ndp\_sink\_100\_0(0)

77.77 Mbps val 2608 name ndp\_sink\_99\_0(0)

78.13 Mbps val 2605 name ndp\_sink\_98\_0(0)

78.13 Mbps val 2602 name ndp\_sink\_97\_0(0)

78.49 Mbps val 2599 name ndp\_sink\_96\_0(0)

78.13 Mbps val 2596 name ndp\_sink\_95\_0(0)

78.13 Mbps val 2593 name ndp\_sink\_94\_0(0)

78.13 Mbps val 2590 name ndp\_sink\_93\_0(0)

78.13 Mbps val 2587 name ndp\_sink\_92\_0(0)

78.13 Mbps val 2584 name ndp\_sink\_91\_0(0)

78.13 Mbps val 2581 name ndp\_sink\_90\_0(0)

78.13 Mbps val 2578 name ndp\_sink\_89\_0(0)

78.13 Mbps val 2575 name ndp\_sink\_88\_0(0)

78.49 Mbps val 2572 name ndp\_sink\_87\_0(0)

77.77 Mbps val 2569 name ndp\_sink\_86\_0(0)

78.13 Mbps val 2566 name ndp\_sink\_85\_0(0)

78.13 Mbps val 2563 name ndp\_sink\_84\_0(0)

78.13 Mbps val 2560 name ndp\_sink\_83\_0(0)

78.49 Mbps val 2557 name ndp\_sink\_82\_0(0)

78.49 Mbps val 2554 name ndp\_sink\_81\_0(0)

78.13 Mbps val 2551 name ndp\_sink\_80\_0(0)

78.13 Mbps val 2548 name ndp\_sink\_79\_0(0)

78.13 Mbps val 2545 name ndp\_sink\_78\_0(0)

78.13 Mbps val 2542 name ndp\_sink\_77\_0(0)

78.13 Mbps val 2539 name ndp\_sink\_76\_0(0)

78.13 Mbps val 2536 name ndp\_sink\_75\_0(0)

78.13 Mbps val 2533 name ndp\_sink\_74\_0(0)

78.13 Mbps val 2530 name ndp\_sink\_73\_0(0)

77.77 Mbps val 2527 name ndp\_sink\_72\_0(0)

78.13 Mbps val 2524 name ndp\_sink\_71\_0(0)

78.13 Mbps val 2521 name ndp\_sink\_70\_0(0)

78.13 Mbps val 2518 name ndp\_sink\_69\_0(0)

78.13 Mbps val 2515 name ndp\_sink\_68\_0(0)

78.13 Mbps val 2512 name ndp\_sink\_67\_0(0)

78.49 Mbps val 2509 name ndp\_sink\_66\_0(0)

78.49 Mbps val 2506 name ndp\_sink\_65\_0(0)

78.13 Mbps val 2503 name ndp\_sink\_64\_0(0)

78.13 Mbps val 2500 name ndp\_sink\_63\_0(0)

78.13 Mbps val 2497 name ndp\_sink\_62\_0(0)

78.13 Mbps val 2494 name ndp\_sink\_61\_0(0)

78.13 Mbps val 2491 name ndp\_sink\_60\_0(0)

78.13 Mbps val 2488 name ndp\_sink\_59\_0(0)

77.77 Mbps val 2485 name ndp\_sink\_58\_0(0)

78.13 Mbps val 2482 name ndp\_sink\_57\_0(0)

78.49 Mbps val 2479 name ndp\_sink\_56\_0(0)

78.13 Mbps val 2476 name ndp\_sink\_55\_0(0)

78.49 Mbps val 2473 name ndp\_sink\_54\_0(0)

78.13 Mbps val 2470 name ndp\_sink\_53\_0(0)

78.13 Mbps val 2467 name ndp\_sink\_52\_0(0)

78.13 Mbps val 2464 name ndp\_sink\_51\_0(0)

78.13 Mbps val 2461 name ndp\_sink\_50\_0(0)

78.49 Mbps val 2458 name ndp\_sink\_49\_0(0)

77.77 Mbps val 2455 name ndp\_sink\_48\_0(0)

78.13 Mbps val 2452 name ndp\_sink\_47\_0(0)

77.77 Mbps val 2449 name ndp\_sink\_46\_0(0)

77.77 Mbps val 2446 name ndp\_sink\_45\_0(0)

78.13 Mbps val 2443 name ndp\_sink\_44\_0(0)

77.77 Mbps val 2440 name ndp\_sink\_43\_0(0)

77.77 Mbps val 2437 name ndp\_sink\_42\_0(0)

77.77 Mbps val 2434 name ndp\_sink\_41\_0(0)

77.77 Mbps val 2431 name ndp\_sink\_40\_0(0)

78.13 Mbps val 2428 name ndp\_sink\_39\_0(0)

78.13 Mbps val 2425 name ndp\_sink\_38\_0(0)

78.13 Mbps val 2422 name ndp\_sink\_37\_0(0)

77.77 Mbps val 2419 name ndp\_sink\_36\_0(0)

77.77 Mbps val 2416 name ndp\_sink\_35\_0(0)

77.77 Mbps val 2413 name ndp\_sink\_34\_0(0)

77.77 Mbps val 2410 name ndp\_sink\_33\_0(0)

78.13 Mbps val 2407 name ndp\_sink\_32\_0(0)

78.13 Mbps val 2404 name ndp\_sink\_31\_0(0)

78.49 Mbps val 2401 name ndp\_sink\_30\_0(0)

77.77 Mbps val 2398 name ndp\_sink\_29\_0(0)

77.77 Mbps val 2395 name ndp\_sink\_28\_0(0)

78.13 Mbps val 2392 name ndp\_sink\_27\_0(0)

77.77 Mbps val 2389 name ndp\_sink\_26\_0(0)

77.77 Mbps val 2386 name ndp\_sink\_25\_0(0)

77.77 Mbps val 2383 name ndp\_sink\_24\_0(0)

77.77 Mbps val 2380 name ndp\_sink\_23\_0(0)

77.77 Mbps val 2377 name ndp\_sink\_22\_0(0)

77.77 Mbps val 2374 name ndp\_sink\_21\_0(0)

77.77 Mbps val 2371 name ndp\_sink\_20\_0(0)

77.77 Mbps val 2368 name ndp\_sink\_19\_0(0)

77.77 Mbps val 2365 name ndp\_sink\_18\_0(0)

77.77 Mbps val 2362 name ndp\_sink\_17\_0(0)

77.77 Mbps val 2359 name ndp\_sink\_16\_0(0)

77.77 Mbps val 2356 name ndp\_sink\_15\_0(0)

77.77 Mbps val 2353 name ndp\_sink\_14\_0(0)

78.13 Mbps val 2350 name ndp\_sink\_13\_0(0)

77.77 Mbps val 2347 name ndp\_sink\_12\_0(0)

77.77 Mbps val 2344 name ndp\_sink\_11\_0(0)

77.77 Mbps val 2341 name ndp\_sink\_10\_0(0)

77.77 Mbps val 2338 name ndp\_sink\_9\_0(0)

78.85 Mbps val 2335 name ndp\_sink\_8\_0(0)

78.13 Mbps val 2332 name ndp\_sink\_7\_0(0)

78.49 Mbps val 2329 name ndp\_sink\_6\_0(0)

78.13 Mbps val 2326 name ndp\_sink\_5\_0(0)

77.77 Mbps val 2323 name ndp\_sink\_4\_0(0)

46.95 Mbps val 2320 name ndp\_sink\_3\_0(0)

78.49 Mbps val 2317 name ndp\_sink\_2\_0(0)

78.49 Mbps val 2314 name ndp\_sink\_1\_0(0)

80.64 Mbps val 2311 name ndp\_sink\_0\_0(0)

Mean of lower 10pc (12 entries) is 9011946.241911 total mean 9690105.462917  mean2 0.000000

对比NDP与TCP吞吐率，NDP吞吐率=1232055000.000000>>TCP吞吐率= 9690105.462917，从这个简单的测试中可以看到NDP协议在吞吐率的表现上是远好于TCP协议的。

## （三）代码分析及功能函数

这里我们来分析NDP吞吐率测试实验。

打开main\_ndp\_permutation.cpp。

为了实验中用到拓扑，包含了一个简单的拓扑头。在大多数实验中，这将会是完全的二等分的胖树拓扑。一系列的定义具体说明了使用的默认参数；节点的数量有定义，数据包中默认的每个端口队列大小也有定义，等等。

eventlist.setEndtime(timeFromSec(0.201));

eventlist是驱动htsim模拟器的类；它是一堆事件，每个都有自己的时间。用setEndTime()设定总的模拟时间。对于ndp实验，这个时间可能会非常小（例如0.2秒）因为ndp汇聚很快；对于tcp而言，需要至少1秒来跑实验，并确保慢启动阈值正确的设定来保证快的汇聚。

然后我们启动logfile，设定它来记录来自模拟器从头开始的事件。

logfile.setStartTime(timeFromSec(0));

NdpSinkLoggerSampling对象将迭代所有的NdpSinks并且每10ms记录他们的速率。

NdpSinkLoggerSampling sinkLogger = NdpSinkLoggerSampling(timeFromMs(10), eventlist);

下面这一行生成胖树拓扑：

FatTreeTopology\* top = new FatTreeTopology(no\_of\_nodes, queuesize,&logfile, &eventlist,ff,COMPOSITE,0);

COMPOSITE参数具体说明使用的队列类型。对于ndp，我们使用组合的队列执行了数据包trimming（源代码在../compositequeue.cpp）

之后，我们生成一条连接模型并且产生一个排列：

ConnectionMatrix\* conns = new ConnectionMatrix(no\_of\_nodes);

conns->setPermutation(no\_of\_conns);

通过在ConnectionMatrix类中使用不同的方法可以产生不同的流量模型包括随机（设定成Random），多对一，多对多等等。

之后，我们生成实际的连接，逐个迭代流量模型生成的连接。生成并启动一个ndp连接的代码如下：

ndpSrc = new NdpSrc(NULL, NULL, eventlist);

...

ndpSnk = new NdpSink(eventlist, 1 /\*pull at line rate\*/);

...

routeout = new Route(\*(net\_paths[src][dest]->at(choice)));

routeout->push\_back(ndpSnk);

routein = new Route(\*top->get\_paths(dest,src)->at(choice));

routein->push\_back(ndpSrc);

ndpSrc->connect(\*routeout, \*routein, \*ndpSnk, timeFromMs(extrastarttime));

连接呼叫需要对外和返回的路由（这是从可用的路由里随机选择的），启动时间也是如此。一般情况下，可能是0或者一个接近于0的小的随机值（为了避免阶段影响）。最后，鉴于路由策略，ndp端点可能需要知道所有可用的线路：

ndpSrc->set\_paths(net\_paths[src][dest]);

ndpSnk->set\_paths(net\_paths[dest][src]);

在所有连接建立后，运行模拟器的主要事件回路：

while (eventlist.doNextEvent()) {

}

当没有待处理事件或者模拟器的时间耗尽时便结束。

## （四）组合传输模型测试

这个测试我们生成一个完全组合的传输模型：每个主机向另一个主机传送数据并接再从另外一个主机接受数据。这个例子使用的是胖树结构完全的横向面带宽，也证明了每个主机能同时发送和接收数据。虽然这可能不是一个最理想的例子，但是能够证明网络是可以在没有大的不公平情况下以100%利用率运行的，甚至有一些主机在发送数据给其他主机时比发送在本拓扑内的主机还要快。

这里我们比较NDP、DCTCP、MPTCP三个协议。

运行本实验模型，需要安装python和gnuplot。

在/sim/EXAMPLES/permutation下，运行“./run.sh”就能运行该实验模型，NDP协议的输出结果在logfile.rates文件当中，DCTCP协议的输出结果在dctcp\_logfile.rates文件当中，MPTCP协议的输出结果在mptcp\_logfile.rates文件当中，数据对比如下图：



总结一下，在这个实验中，我们模拟了432个以接近10Gb/s速率发送数据的节点，所以聚合负载接近4Tb/s。即使是9KB的数据包，也需要花几分钟时间来运行。对于NDP而言，队列大小是8个数据包，并且我们使用23个数据包的初始窗口。23个数据包对于NDP来说是一个大的初始窗口，但对于全双工传输，我们始终产生小的队列，并且PULLs也会有排队的发生。NDP对初始窗口大小的选择并不敏感，但是在这种情况下，更小的初始窗口值将不会有足够的线上数据包来满足网络。在真实的负载下，我们将永远不需要这么大的初始窗口。

DCTCP在胖树拓扑的这个组合传输模型下表现的比较差，因为它对于每个流只使用单一的路线，路由则是使用一个采取了ECMP的随机路线，并且结果就是遭遇了流冲突。为了让它表现的尽可能好，DCTCP也需要比NDP更多的排队，这当中我们使用的队列长度达100，增加超过100并不能进一步提高性能。一般而言，单一路线策略在更高的负载下不太能具有鲁棒性。

## （五）其他的实验

运行incast和其他实验，工作流程近乎相似。运行htsim\_\*的可执行文件并且提供需要的参数。htsim代码包含TCP NewReno (not SACK)和一些MPTCP协议（包含各种拥塞控制算法）的实施。DCTCP,PFC/DCQCN也支持（DCQCN是基于DCTCP代码并且是基于窗口而不是速率）。为了跑这些实验可以使用适当的实验文件，或者可以重新定义。

## 四、总结

通过此次实验，在测试环境下，我们可以看到NDP相对于现有其他协议的优秀表现，但是它在一些其他的环境下，表现出来的性能也有一定的优劣。

好的表现环境包括：1.同时的incast和outcast模型；2.多样的数据包大小和流模型；3.过载的拓扑环境；4.有管理程序支持的虚拟机器。

不好的表现环境包括： 1.不均等长度的多路径情况，例如Bcube，在没有每条路线速率控制的情况下，可能会表现的非常不公平；2.公用因特网。