k元的fat tree， 特征：
- 每台交换机都有k个端口
- 核心层为顶层，一共有$\left(\frac{k}{2}\right)^2$个交换机
- 一共有k个pod，每个pod有k台交换机组成，汇聚层和接入层各占$\frac{k}{2}$个交换机
- 接入层每个交换机可容纳k/2台服务器。k元fat-tree一共有k个pod，每个pod容纳$\left(\frac{k}{2}\right)^2$个服务器，所有pod容纳$\boldsymbol{k}\left(\frac{k}{2}\right)^2$个服务器
- 任意两个pod之间存在 k 条路径

switch 标号：
- 网络中的地址10.0.0.0/8
- pod中的地址：`10.pod.switch.1`。$pod \in [0, k-1]$
  $switch \in [0, k-1]$ 从左下角0，到右上角k-1（port的编号同理）
- Core的地址：`10.k.j.i`，k为k元，固定值。j,i，为交换机的坐标。$j, i \in [1, \frac{k}{2}]$，从左上角第一个开始起

路由算法：（前缀匹配路由表，后缀匹配路由表）
- 向上使用两层路由，向下使用一层前缀匹配路由

Prefix	Output port
10.2.0.0/24	0
10.2.1.0/24	1
0.0.0.0/0	

Suffix	Output port
0.0.0.2/8	2
0.0.0.3/8	3

聚合层路由算法：
- 3~5行：每一个边缘层switch的路由表
- 6~9行：聚合层向外输出到核心层，根据(i-2+z)mod(k/2)+(k/2)计算出的端口可以将跨pod访问指定主机的信息从正确的端口转发出去，模运算的目的是保证**负载均衡**。

边缘层路由算法：
- switch z的范围改为[0,k/2-1]，无需向下路由（忽略3~5行）

核心层路由算法：
- 3~5行：对指定的pod，从指定的端口转发出去
- Core switch的路由表：(10.pod.0.0/16, pod)

匹配的是目的地址

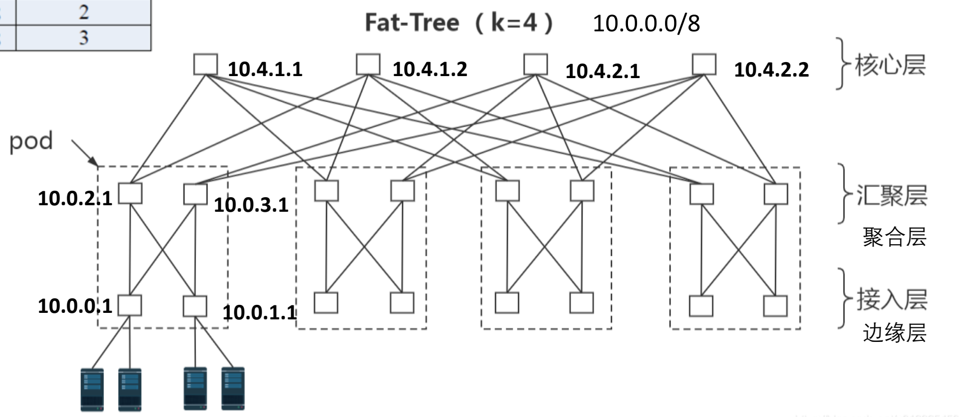
addPrefix(switch, address, port)给switch添加address的ouput port为port

```
1  foreach pod $x$ in $[0, k-1]$ do    对所有pod循环
2      foreach switch $z$ in $[(k/2), k-1]$ do  对汇聚层的switch循环
3          foreach subnet $i$ in $[0, (k/2)-1]$ do 对边缘层的switch循环
4              addPrefix($10.x.z.1$, $10.x.i.0/24$, $i$);
5          end   将第i个边缘层switch连接到该聚合层switch的输出port i
6          addPrefix($10.x.z.1$, 0.0.0.0/0, 0);
7          foreach host ID $i$ in $[2, (k/2)+1]$ do
8              addSuffix($10.x.z.1$, $0.0.0.i/8$,
               $(i-2+z) mod (k/2) + (k/2)$);
9          end
10     end
11 end
```

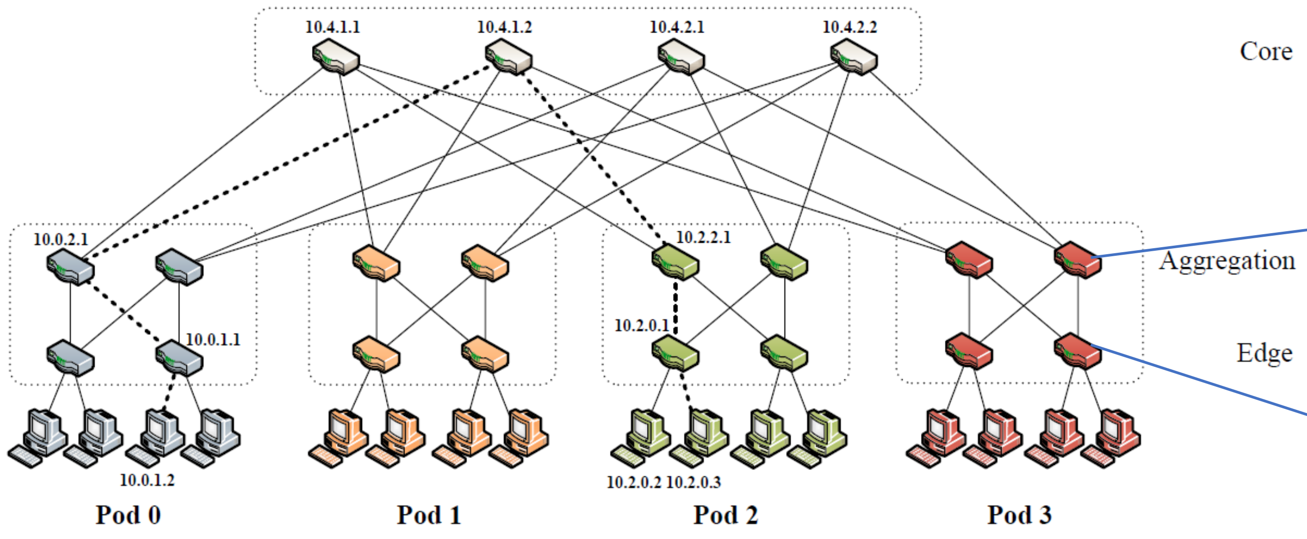
聚合层路由算法

```
1 foreach $j$ in $[1, (k/2)]$ do
2     foreach $i$ in $[1, (k/2)]$ do
3         foreach destination pod $x$ in $[0, (k/2)-1]$ do
4             addPrefix($10.k.j.i$,$10.x.0.0/16$, x);
5         end
6     end
7 end
```

Algorithm 2: Generating core switch routing tables.



从Pod0的10.0.1.2发送消息到10.2.0.3
查询目的地址



向上有prefix和suffix表
向下有prefix路由表

向上只有suffix表
向下用标准路由
表

向上匹配
- 10.0.1.2 to 10.0.1.1：由标准路由完成，因为10.0.1.1是主机网管。
- 10.0.1.1 to 10.0.2.1：查前缀路由表进入0.0.0.0/0的二级后缀路由表查到0.0.0.3/8在Port 2出（由代码6~9行表明,i=3,z=1，算出port=2,端口0、1映射下面主机，端口2、3映射聚合层Switch，从左往右2表示10.0.2.1）。
- 10.0.2.1 to 10.4.1.2：查前缀路由表进入0.0.0.0/0的二级后缀路由表查到0.0.0.3/8在Port 3出（由代码6~9行表明,i=3,z=2，算出port=3,端口0、1映射下面edge的Switch，端口2、3映射Core Switch第0行第0、1个，从左往右3表示10.4.1.2）。

向下匹配
- 10.4.1.2 to 10.2.2.1: 由前缀路由表匹配的到Pod值为2，发送到第2个Pod;由于j=1，故发送给Pod2中第1个Switch 10.2.2.1
- 10.2.2.1 to 10.2.0.1: 由前缀表匹配到（代码3~5行中i=0），指向第0个Edge层Switch 10.2.0.1
- 10.2.0.1 to 10.2.0.3:标准路由技术

10.0.0.0/16	0
10.1.0.0/16	1
10.2.0.0/16	2
10.3.0.0/16	3

Core Switch的路由表

MPTCP拥塞控制的目标：(Multipath TCP)

- 确保MPTCP的引入不会对原有网络流量造成严重干扰(公平性等)
- 尽可能地保证高吞吐率

理解EWTCP和Coupled

- MPTCP协议本质是作为一个传输层的中间件出现的，它的主要目的是将一个数据流划分到多个路径上传输。一个TCP连接管理多个subflow，每个subflow可以在网络中进行自由选路，并且每一个subflow都维护着属于自己的一个私有的发送窗口Wr，而MPTCP的发送者则在任何一个subflow发送窗口有余量时将数据分发给它们去传输，以上是对MPTCP的一个简要概括

公平性：一个非常朴素的思想就是给每个流可以获得的带宽进行一个加权限制，进而引出了EWTCP(Equally-Weighted TCP)拥塞控制算法

- 如果直接将原始的TCP拥塞控制用在MPTCP场景中，那么MPTCP的每一个subflow都将以独立的竞争者身份和原始的单路径TCP流进行竞争，这样显然对单路径TCP是非常不利的。可以想象，假设在RTT相同的情况下，一个拥有n个subflow的MPTCP，其最终获取的带宽将会是单路径TCP的n倍，极大地损害了网络的公平性，不能实现与现有TCP流量的和谐共存。
- 如果取a = $\frac{1}{\sqrt{n}}$，并且假设RTT值都相等，就会使得最终含有n个subflow的MPTCP作为一个整体可以在瓶颈链路上获得和一个单路径TCP同样的网络带宽，从而强行地获得了分配上的公平性。
- 当n = 1时，EWTCP会退化到Regular TCP拥塞控制算法。此外，EWTCP算法中涉及到的MPTCP每一个subflow彼此之间相互独立(因为是按照Wr 来进行拥塞控制的)，所以即使有一条subflow被阻塞了，其他的subflow也不会因此变得更加有侵略性，这保证了MPTCP实体不可能在总占用带宽上超过单路径TCP。

ALGORITHM: EWTCP

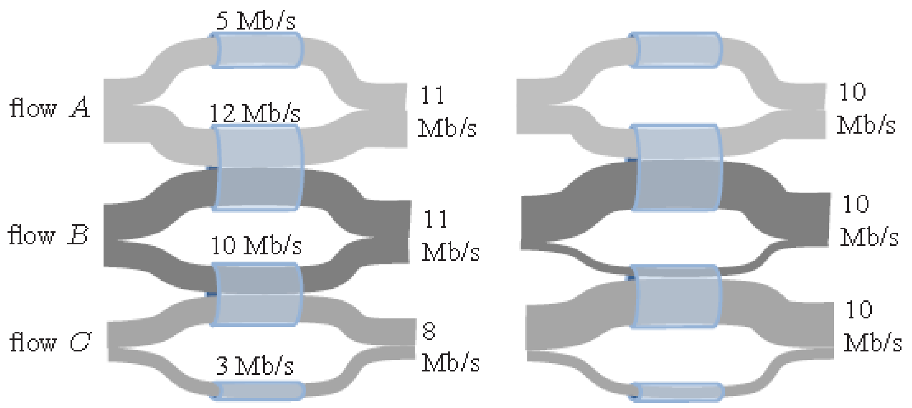
- For each ACK on path r, increase window w_r by a/w_r.
- For each loss on path r, decrease window w_r by $w_r/2$.

Here w_r is the window size on path r, and $a = 1/\sqrt{n}$ where n is the number of paths.

ALGORITHM: COUPLED

- For each ACK on path r, increase window w_r by $1/w_{\text{total}}$.
- For each loss on path r, decrease window w_r by $w_{\text{total}}/2$.

Here w_{total} is the total window size across all subflows.



不需要进行拥塞程度侦测也可以自适应地找到最优解，这就是COUPLED算法：

- 其中W_{total} 是一个MPTCP实体下管辖的所有subflow的发送窗口大小之和。要理解这个协议需要分两种情况：

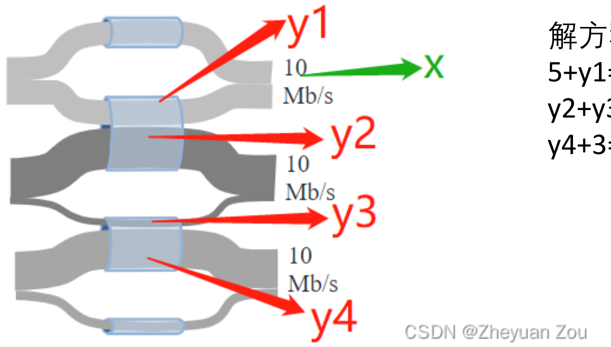
1. 各个链路拥塞程度相等的情况
   每一个subflow的发送窗口增长和发送窗口减少，从长远来看一定要相互抵消，就是解下面这个方程(p是丢包率，1-p是成功发送概率)解出来发现 $W_{total} = \sqrt{2(1-p)(p)} \approx \sqrt{\frac{2}{p}}$，这直接指向了另外一个重要结论：在COUPLED算法中，一个MPTCP的各个subflow可以获得的**总带宽之和只与链路丢包率有关**，这证明了COUPLED拥有天然的带宽竞争公平性，这是各个链路丢包率相同时的情况

$$\left(\frac{w_r}{\mathrm{RTT}}(1-p)\right)\frac{1}{w_{\mathrm{total}}} = \left(\frac{w_r}{\mathrm{RTT}}p\right)\frac{w_{\mathrm{total}}}{2}.$$

2. 各个链路拥塞程度不同的情况
   首先注意COUPLED算法中对每一条subflow而言，每次增加或减少的窗口值对所有subflow是统一的，只与那个时刻的W_{total}有关。那么拥塞程度高的链路势必在指定的一段时间内会遇到更多的丢包，长远来看最终算法会慢慢地将其流量逐渐向低拥塞链路汇集，从而实现了EWTCP没有实现自适应功能。

EWTCP的计算：
它会倾向于友好地和每一个竞争者保持均等分配。所以在上面左图中，12Mb/s的链路和10Mb/s的链路都会被均分。
所以A=5+6=11, B=6+5=11, C=5+3=8

COUPLED的计算：
各个MPTCP实体会获得相等的全局链路带宽，所以它们最终会获得的带宽是一致的，设为统一的x。设以下未知量：



解方程组：
5+y1=10
y2+y3=10
y4+3=10

首先BBR的全称是(Bottleneck Bandwidth and Round-trip propagation time)，基于瓶颈链路带宽和往返传播时延的拥塞控制协议。BBR的动作要义有两个：一是根据BDP来控制链路中inflight的流量，二是根据瓶颈链路带宽来控制数据包发送节奏，使其不断适应BtlBW的变化

瓶颈链路，顾名思义，指的是在整个通信链路上分得带宽最小的那一截链路，它是整条通信链路中源-目标吞吐量的决定性因素，如果发生拥塞，在瓶颈链路这里将最先发生数据包的缓存，它的带宽叫做瓶颈链路带宽(BtlBW)。

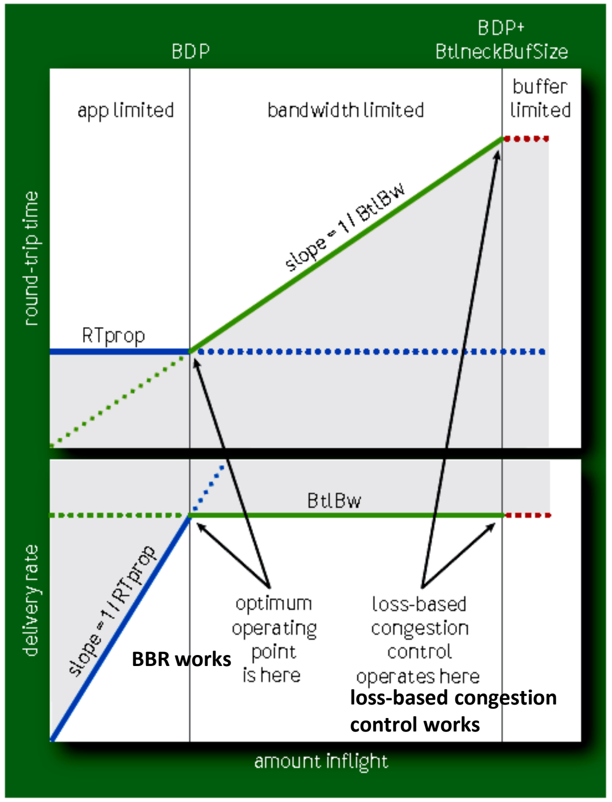
往返传播时延(RTprop)指的是网络完全轻载(没有排队时间等)时的往返传播延迟之和。这两者的乘积被定义为 $\mathrm{BDP} = \mathrm{BtlBW} \times \mathrm{RTprop}$ 可以想见BDP的含义就是在网络不产生拥塞的情况下，积攒(inflight)在整个链路中的数据量的最大值。

应用受限阶段：传输delay不变，bandwidth增长，直到达到BDP
- 在应用受限阶段，网络链路处于轻载状态，能发多快全看应用想发多快，此时的往返延迟就正好等于RTprop没有排队时间，因为轻载)。而交付速率(单位时间内成功发送的分组数量)也和应用端向网络中注入的inflight流量呈正相关性，斜率为 $\frac{1}{\mathrm{RTprop}}$

带宽受限阶段：packet starts to queue。bandwidth不变，传输delay增长
- 随着应用注入数据的速度不断加快，网络就开始从应用受限阶段进入到带宽受限阶段。注意两者过渡点的inflight流量恰好是BDP，**这是一个应该开始进行拥塞控制的最优动作点（BBR works）**。经过此点之后将进入带宽受限阶段，这个阶段将开始有数据被缓冲在瓶颈链路的buffer中。这个阶段的交付速率也将被严格地限制为BtlBW，受限因素是瓶颈链路。此时往返延迟也开始增加(因为开始有排队时间)，斜率(也就是释放速度)为 $\frac{1}{\mathrm{BtlBW}}$

缓存受限阶段：packet starts to be dropped
- 注意从带宽受限阶段到缓存受限阶段的过渡点，其inflight流量是BDP+BtlBufSize，意思是此时瓶颈链路上的缓冲区已经充满，再继续注入新的流量就会发生缓冲区溢出而丢包。所以在进入缓存受限阶段之后，因为有丢包的可能存在，其交付速率和往返时间都变得不可估算了

FIGURE 1: **DELIVERY RATE AND ROUND-TRIP TIME VS. INFLIGHT**



Loss-based congestion control
- Operate at the right edge of bandwidth limited region
- 代价是高延迟和频繁丢包

Elastic Sketch:基于 sketch（概述）的网络测量：哈希表、CM、bloom filter

Sketches

•Hash table

•Count-min (CM)

Elastic

Heavy part（少量的大流量）使用 hash table，light part（大量的小流量）使用 CM

插入流时在哈希表中检查：

•没有 ⇒ 直接插入

•存在，且 bucket 的流一致 ⇒ vote+ 自增

•存在，但是流不一致，且 vote- 没有优势：vote- 自增，在 CM 中插入 (f, 1)

•存在，流不一致，vote- 有优势：vote- 自增，把 bucket 里面的流赶到 CM 里面，特别地设置 bucket 里面 flag = T（flag 表示 light part 里面是否有这个流的统计结果）

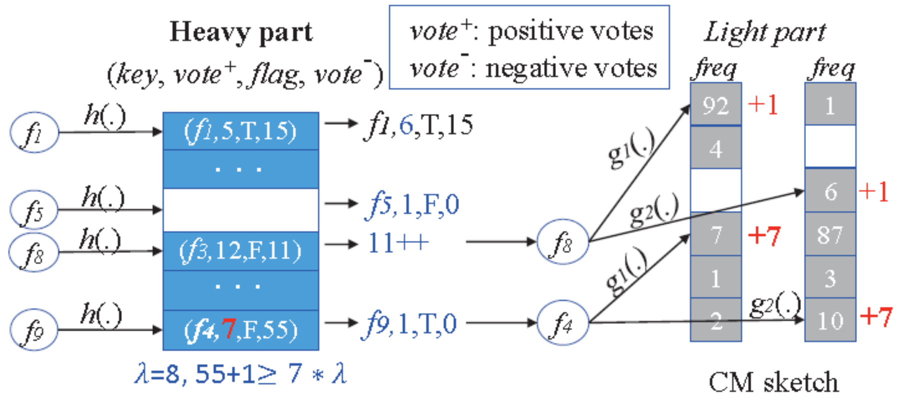
查询：

•f 不在 heavy part 里面：用 CM 估计

•f 在 heavy part 里面：flag = F ⇒ 返回 vote+；flag = T ⇒ 返回 vote+ 与 CM 结果的和

数据结构

- Heavy part for elephant flows
  - A hash table *H*
  - Each hash bucket contains:
    - flow ID, e.g., 5-tuple
    - Positive votes ($vote^+$): number of packets belong to the flow
    - Negative votes ($vote^-$): number of other packets
    - flag: whether light part contains positive votes for this flow.
- Light part for mouse flows
  - A CM sketch



插入

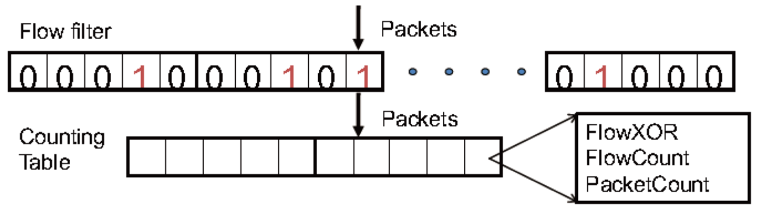
- On receiving a packet of flow f
  - Hash to H[h(f)%B], suppose the bucket is (f_1, $vote^+$, $flag_1$, $vote^-$)
  - Case 1: the bucket is empty, insert (f, 1, F, 0) into it.
  - Case 2: $f=f_1$, increment $vote^+$ by 1
  - Case 3: $f\neq f_1$, and $vote^-/ vote^+ < \lambda$, increment $vote^-$ by 1, and insert (f,1) to CM by incrementing the counters by 1.
  - Case 4: $f\neq f_1$, and $vote^-/ vote^+ \geq \lambda$ after incrementing $vote^-$ by 1. Evict f by setting the bucket as (f, 1, T, 1), and evict f_1 by incrementing the mapped counter in CM by $vote^+$

查询

- Flow *f* not in heavy part, estimate its size with the CM sketch
- Flow *f* in heavy part,
  - Flag is F: return $vote^+$
  - Flag is T: return sum of $vote^+$ and the query result of CM

Encoded Flowset

- Encode flowset data structure
  - A flow filter: A standard Bloom filter of a bit array, m_f bits and k_f hash functions
  - A counting table: store flow counters, an extended Bloom filter with m_c cells and k_c hash functions, each cell has three fields
    - *FlowXOR*: XOR of all the IDs of the flows mapped to this cell
    - *FlowCount*: number of flows mapped to this cell
    - *PacketCount*: number of packets of the flows mapped to this cell



Packet Processing (Encode)

- Receiving a pacekt, first check whether the flow has been stored in the flowset with flow filter
- If new flow, update the counting table
  - XOR the flow ID to FlowXOR
  - Increment FlowCount and PacketCount
- If existing flow
  - Increment PacketCount

Algorithm 1: FlowRadar packet processing

```
if $\exists\, i \in [1,k_f]$, s.t. $FlowFilter[H_i^F(p.flow)]==0$ then
    FlowFilter.add(p.flow);
    for $j = 1..k_c$ do
        l = $H_j^C$(p.flow);
        CountTable[l].FlowXOR =
        CountTable[l].FlowXOR $\oplus$ p.flow;
        CountTable[l].FlowCount ++;
    end
end
for $j = 1..k_c$ do
    CountTable[$H_j^C$(p.flow)].PacketCount ++;
end
```

SingleDecode

- Switch reports flowset to collector every a few milliseconds
- First look for cells with just one flow.
  - FlowCount==1
- Perform hash functions on a decoded flow to locate other cells, remove it from the cell, decode more flows
- Example
  - Cell[$h_1(f_1)$]: f_1, 1, 10
  - Cell[$h_2(f_1)$]=Cell[$h_1(f_2)$]: f_1 XOR f_2, 2, 25]
  - Remove f_1 from Cell[$h_2(f_1)$], then decode f_2, f_2 has 15 packets.

PIFO

- Only needs two components
  - The push-in first-out (PIFO) queue: a priority queue that allows elements to be enqueued into an arbitrary position based on the element's rank, but dequeues elements from the head.
    - Elements with a lower rank are dequeued first.
  - Packet transaction: the computation of an element's rank before it is enqueued into a PIFO.
    - Scheduling transaction, scheduling tree

	FIFO	PIFO
Packet enqueue	Add to the tail	Insert to arbitrary position based on packet's rank
Packet arrive to a full queue	Drop the packet	Drop the largest ranked packet its rank is larger than the arriving packet
Packet dequeue	Remove from head	Remove from head

SP-FIFO （Strict priority)

- A port has multiple FIFO queues, each queue is associated with a priority
- A queue starts to transmit packets only when all the queues with higher priorities are empty

算法:

- Receive a packet p with rank r(p)
- Push-up: The mapping process scans the queues bottom-up and enqueue the packet in the first queue that satisfies **r(p)≥qi**. It then increases qi to the rank of the enqueued packet, i.e., qi=r(p).
- Push-down: When detecting inversion, decrease all queue bounds by **q1-r(p) no queue to join**

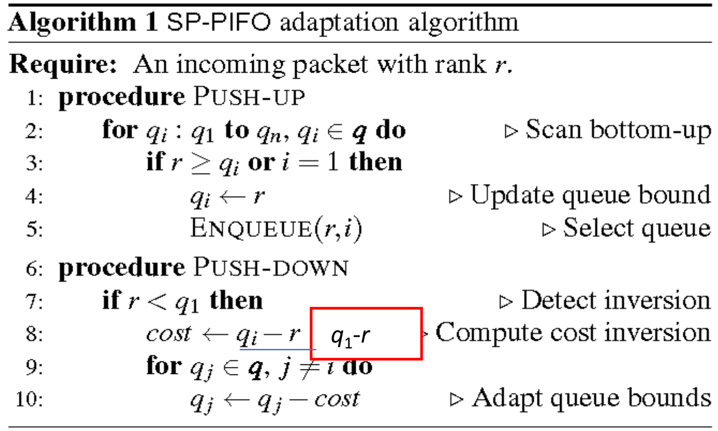
STFQ(Start-time Fair queueing)

- Before a packet is enqueued
- When a packet is dequeued

virtual_time += p.length

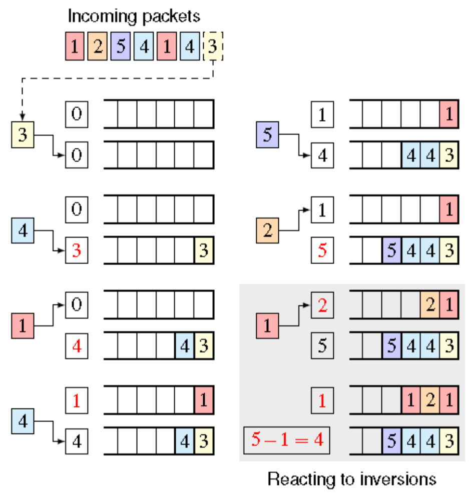
f.weight 流量权重

```
if f in last_finish:
  p.start = max(virtual_time, last_finish[f])
else:
  p.start = virtual_time
last_finish[f] = p.start + p.length / f.weight
p.rank = p.start
```



Algorithm 1 SP-PIFO adaptation algorithm

```
Require: An incoming packet with rank $r$.
 1: procedure PUSH-UP
 2:     for $q_i : q_1$ to $q_n$, $q_i \in \boldsymbol{q}$ do        ▷ Scan bottom-up
 3:         if $r \geq q_i$ or $i = 1$ then
 4:             $q_i \leftarrow r$                  ▷ Update queue bound
 5:             ENQUEUE($r$,$i$)                 ▷ Select queue
 6: procedure PUSH-DOWN
 7:     if $r < q_1$ then                         ▷ Detect inversion
 8:         $cost \leftarrow q_i - r$   $q_1$-r      ▷ Compute cost inversion
 9:         for $q_j \in \boldsymbol{q}$, $j \neq i$ do
10:             $q_j \leftarrow q_j - cost$          ▷ Adapt queue bounds
```

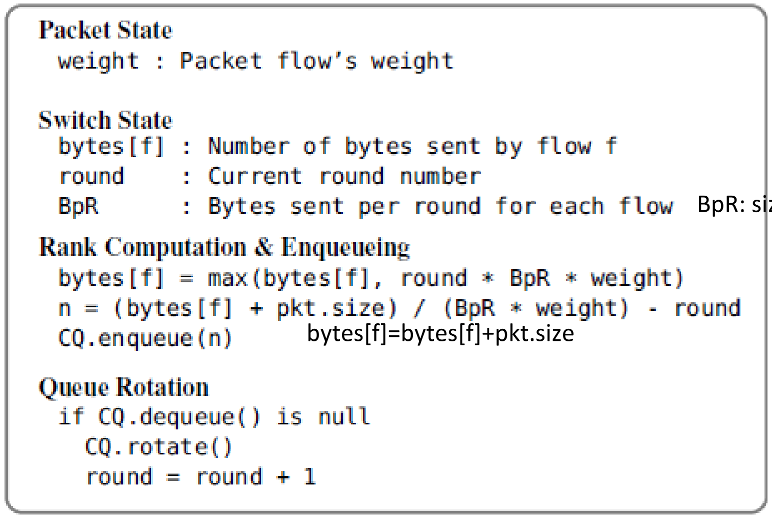


- A PCQ has a fixed number of buckets or FIFO queues, say N, each of which stores packets scheduled for next N periods
- A PCQ supports three methods
  - CQ.enqueue(n): Used by the ingress pipeline to schedule the current packet n periods into the future.
  - CQ.dequeue(): Used by the egress pipeline to obtain a buffered packet, if any, for the current period.
  - CQ.rotate(): Used by the pipelines to advance the CQ so that it can start transmitting packets for the next period.

- Physical Calendar Queue: the CQ moves onto the next queue after a fixed time interval periodically
- Logical Calendar Queue: the CQ advances to the next queue whenever the current queue is empty

WFQ：Weighted fair queueing

- WFQ: n flows, flow fi has a weight wi, and is assigned with a bandwidth of B× wi /Σwj
- Weighted Fair Queueing (WFQ) scheduling achieves max-min fair allocation
- Apply logical calendar queues



Packet State
```
 weight : Packet flow's weight
```

Switch State
```
 bytes[f] : Number of bytes sent by flow f
 round    : Current round number
 BpR      : Bytes sent per round for each flow
```
BpR: size of a queue

Rank Computation & Enqueueing
```
 bytes[f] = max(bytes[f], round * BpR * weight)
 n = (bytes[f] + pkt.size) / (BpR * weight) - round
 CQ.enqueue(n)
```
bytes[f]=bytes[f]+pkt.size

Queue Rotation
```
 if CQ.dequeue() is null
   CQ.rotate()
   round = round + 1
```

Flow1	Flow2	Flow3
1/1.2=0.83→queue 1	1/1.6=0.626→queue 1	1/1.2=0.83→queue 1
2/1.2=1.67→queue 2	2/1.6=1.25→queue 2	2/1.2=1.67→queue 2
3/1.2=2.5→drop	3/1.6→1.875→queue 2	3/1.2=2.5→drop
	4/1.6→2.5→drop	

Example:

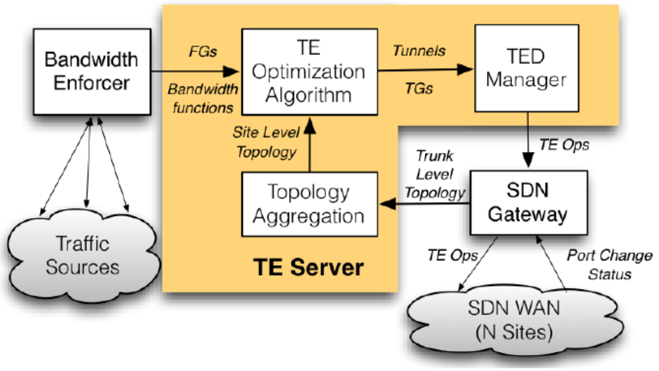
Flow 1, weight=0.3;
Flow 2, weight=0.4;
Flow 3, weight=0.3
Two queues, BpR=4
At time 0, 3 packets of flow1, 4 packets of flow2, 3 packets of flow3 arrives

B4 TE 算法

TE server operates on:

- Network Topology: a graph represents sites as vertices and site to site connectivity as edges.
- Flow Group (FG): a {source site, dest site, QoS} tuple
- A Tunnel (T) represents a site-level path
- A Tunnel Group (TG) maps FGs to a set of tunnels and corresponding weights.

Output Tunnel Groups to SDN gateway, gateway forwards Tunnels and Flow Groups to OFC and install on Switch using OpenFlow



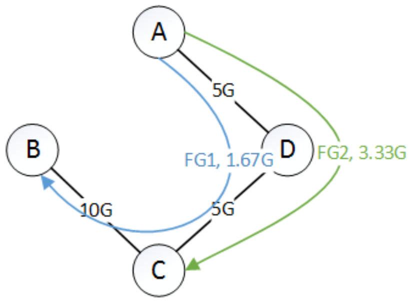
Bandwidth Function

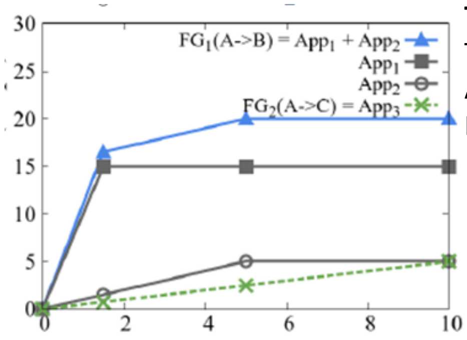
- Application's bandwidth function
  - bandwidth allocation to an application given the flow's relative priority (called fair share)
- FG's bandwidth function
  - Piecewise linear additive composition of per-application bandwidth functions.

A greedy algorithm

- allocates edge capacity among FGs according to their bandwidth function such that all competing FGs on an edge either receive equal fair share or fully satisfy their demand.
- iterates by finding the bottleneck edge when filling all FGs together by increasing their fair share on their preferred tunnel.
- freeze all tunnels cross the bottleneck edge, move to the next preferred tunnel.

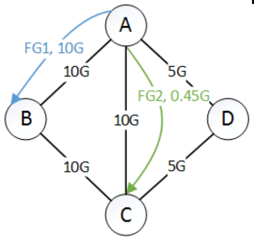
T1,3=A→D → C → B, T2,2=A → D → C, at share 10, A → D and D → C become bottleneck.
At fair share 10, FG2 is allocated 5G, while FG1 is allocated 20G and its requirement is fully satisfied



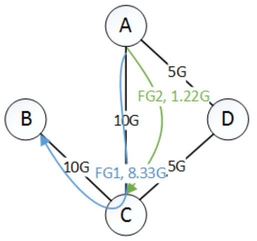


Tx,y: yth most preferred tunnel for FGx.
T1,1=A→B, T2,1=A→C, at share 0.90.
A→B becomes bottleneck
FG1 has 10G, FG2 has 0.45G

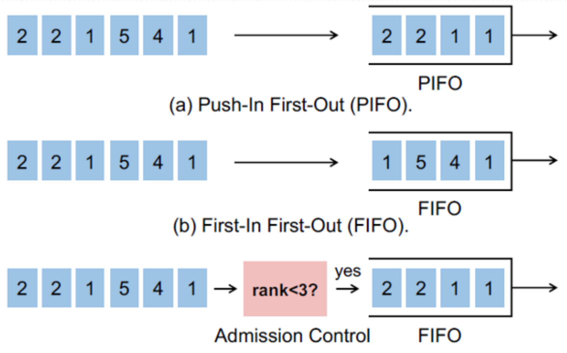
App1 demands 15G A→B
App2 demands 5G A→B
App3 demands 10G A→C
FG1=App1+App2
FG2=App3



T1,2=A→C→B, at share 3.33,
A→C becomes bottleneck
FG1 has 18.33G, FG2 has 1.67G



- Shallow queue: queue size is 4
- PIFO: packets ranked 4 and 5 are evicted by packets ranked 2
- FIFO: two packets ranked 2 are dropped
- AIFO: an oracle admission control proactively drop all packets with ranks no smaller than 3
  - Given fast-converging congestion control, dropped packets will be quickly retransmitted



(a) Push-In First-Out (PIFO).

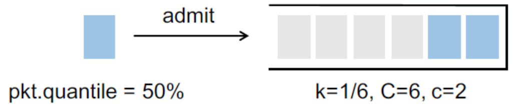
(b) First-In First-Out (FIFO).

(c) Admission-In First-Out (AIFO).

```
Algorithm 1 AIFO
function INGRESS($pkt$)
    // Admission Control
    Update sliding window $W$ with $pkt$
    $c \leftarrow Queue.length$
    $C \leftarrow Queue.size$
    if $c \leq k \cdot C \parallel W.quantile(pkt) \leq \frac{1}{1-k}\frac{C-c}{C}$ then
        // Admit packet
        Queue.enqueue(pkt)
    else
        // Drop packet
        Drop $pkt$
function EGRESS
    if $Queue$ is not empty then
        $pkt \leftarrow Queue.deque()$
        Send $pkt$
```

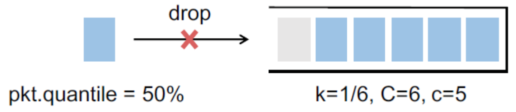
Example： C=6, k=1/6

$$threshold = \frac{C-c}{(1-k)C} = \frac{6-2}{5} = 80\%$$



(a) Admit packet when current queue length c = 2.

$$threshold = \frac{C-c}{(1-k)C} = \frac{6-5}{5} = 20\%$$



(b) Drop packet when current queue length c = 5.

AIFO算法：

At the ingress

- Decide whether to enqueue or drop a packet.
- The threshold is dynamically determined by **queue length (c)** and **queue size (C),** and use **quantile estimation (W.quantile(pkt))** to estimate the relative rank of current packet.
- The queue is a FIFO queue which enqueues the packet to the end of the queue.

At the egress

- When the queue is not empty, AIFO dequeues a packet from the head of the queue, and sends the packet out.

$$c \leq k \cdot C \parallel W.quantile(pkt) \leq \frac{1}{1-k}\frac{C-c}{C}$$

- W.quantile(pkt) estimates the quantile of pkt
- AIFO maintains a sliding window W of recently received packets and uses the quantile of the rank of the arrival packet (W.quantile(pkt)) as the criteria.
- If larger than $\frac{1}{1-k}\frac{C-c}{C}$ drop; other wise admit

More packet admitted (c increases), more difficult to admit packets ((C-c) decreases).

HCSFQ

Hierarchical Fair Queueing

- Flows are grouped into flow aggregates in multiple layers. The root of the tree includes all the flows.
- Each node in the tree includes a subset of the flows, called a flow aggregate, and fairly allocates its bandwidth to its child nodes.
- This is done recursively until leaf nodes, each of which contains one flow.
- The key benefit of hierarchical fair queueing is that it allows unused share of a flow to be allocated to other flows in the same flow aggregate, instead of being shared by all the flows.

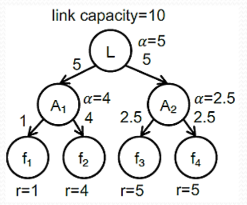
算法

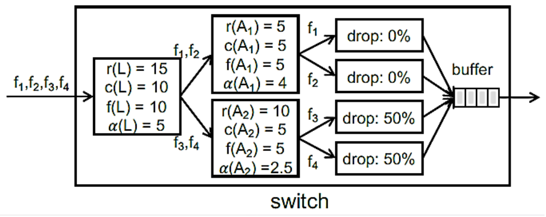
- Update $r(v)$ with exponential averaging each time a new packet of v arrives
- A packet carries
  - $pkt.r$ is the arrival rate of the flow the packet belongs to
  - $pkt.node$ is a list of node IDs indicating the flow aggregates the flow belongs to, e.g., $pkt.node$=[L,A_1]

1. After receiving a packet, update $r(v)$ for all the nodes the packet belongs to
2. At the root node v, $c(v) = C$,
   a) If $r(v) > c(v)$, compute the fair share rate $\alpha(v)$ by solving $c(v) = \sum_u \min(r(u), \alpha(v))$
   b) Else, compute the fair share rate $\alpha(v)$ as $\alpha(v) = \max_u(r(u))$
3. For each child node u of v, $c(u) = \min(r(u), \alpha(v))$.
4. Repeat 1-3 until reach to leaf nodes
5. Compute packet dropping probability as
$$\max(0, 1 - \frac{\alpha(v.parent)}{r(v)})$$

Example

- r(L)=15, r(A1)=5, r(A2)=10, r(f1)=1, r(f2)=4, r(f3)=r(f4)=5
- At root, obtain $\alpha(L)$ by solving
  $C(L) = 10 = \min(5, \alpha(L)) + \min(10, \alpha(L))$, $\alpha(L) = 5$, $C(A_1) = \min(5,5) = 5$, $C(A_2) = \min(5,10) = 5$
- At A1, obtain $\alpha(A_1)$ by $\alpha(A_1) = \max(1,4) = 4$
- At A2, obtain $\alpha(A_2)$ by solving
  $5 = \min(5, \alpha(A_2)) + \min(5, \alpha(A_2))$, $\alpha(A_2) = 2.5$
- f1 and f2 have drop prob. of 0; f3 and f4 have drop prob. Of 0.5





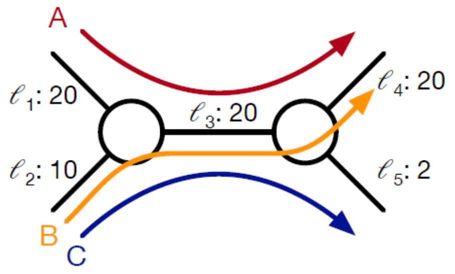
Weighted HCSFQ

- Flows have weights
- Two changes
  - Change $c(v) = \sum_u \min(r(u), \alpha(v))$ to
$$c(v) = \sum_u w(u) \cdot \min(\frac{r(u)}{w(u)}, \alpha(v))$$
  - Change the drop prob. as
$$\max(0, 1 - \alpha(v.parent) \cdot \frac{w(v)}{r(v)})$$

Water filling algorithm

- Max-min fairness can be achieved by iterative water-filling algorithm
  - Intuitively, the algorithm initializes all flows as 'unconstrained' with a rate of 0.
  - In every iteration, the algorithm adds an equal amount to all unconstrained flows until at least one link in the network becomes saturated.
  - All flows that traverse the saturated link are now considered 'constrained.'
  - The algorithm iterates until all flows are constrained.

- First iteration, flow A=flow B=flow C=2, flow C is bottlenecked by l_5, stop increasing rate
- Second iteration, flow A=flow B=8, flow C=2, flow B is bottlenecked by l_2, stop increasing rate
- Third iteration, flow A=10, flow A is bottlenecked by l_3



(b) Multiple bottlenecks

Max-min fairness

- **Definition 1**. Let R be the set of all possible flow-rate allocations that satisfy the capacity constraints of the network. An allocation of rates $\vec{r} = \{r_1, \dots, r_n\}$ in R is "max-min fair" if and only if, for all other allocations $\vec{s} \in R$ and all flows i:
  $s_i > r_i \rightarrow \exists j: \left(r_j \leq r_i \; and \; s_j < r_j\right)$
  In other words, there exists in the other allocations, $\vec{s}$, a smaller flow, j, that loses capacity.
- Traditional TCP congestion avoidance share many similarities to the water-filling algorithm, but later diverges in numerous and significant ways.
  - Flows are not simultaneously initialized with zero rates,
  - Rate increments are not simultaneous nor uniform (depends on RTTs)
  - Heterogeneous congestion detection methods (e.g., loss, delay, ECN, hybrid, etc.) and increase/decrease algorithms

Cebinae Approach

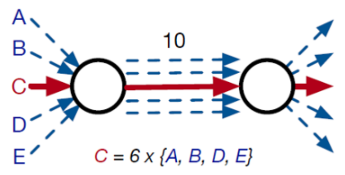
- **Definition 2**. An allocation of rates $\vec{r} = \{r_1, \dots, r_n\}$ in R is "max-min fair" if and only if, for all flows r_i , there exists at least one bottleneck link for r_i , l, that satisfies both the following properties:
  - l is saturated. Specifically, $capacity_l = \sum_{j \in L} r_j$, where L is the set of all flows utilizing l.
  - r_i is the largest flow. Specifically, $\forall j \in L : (r_i \geq r_j)$.

- Definition 2 lends itself to an efficient and distributed verification of the max-min fairness of a network.

1. If the link is not saturated, it is not a bottleneck for any flow currently using the link.
2. If the link is saturated, then for each flow i on the link:
   a) If i has the largest rate among the locally competing flows, then this link is i's bottleneck. i can only capture more bandwidth at the cost of other flows. Note that multiple flows can be bottlenecked by the same link if their rates are equal.
   b) If i does not have the largest rate on the link, then the link is not i's bottleneck. i may or may not have a bottleneck link elsewhere in the network.

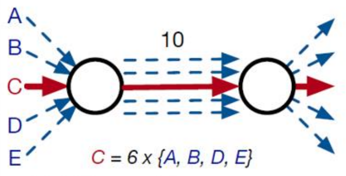
Example

- Cebinae will rate-limit flow C to a rate of $6(1-\tau)$.
- The other flows are then allowed to reclaim the taxed capacity.
- the network will converge to max-min fairness in $\frac{\ln(\frac{1}{3})}{\ln(1-\tau)}$ timesteps

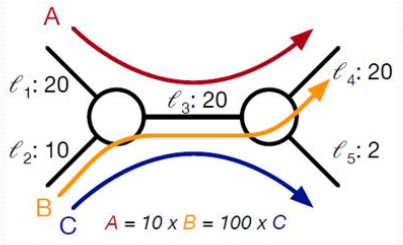


Limitations of Strawman Solution

- Limitation 1: Cannot make an already-unfair allocation fair.
  - Converged allocation of {1,1,6,1,1}
  - Correctly limit flow C's bandwidth because it has already achieved its fair share.
  - Other flows do not have a mechanism to claim their own fair share.



- Limitation 2: Flow C dose not responsive to a simple token-bucket filter, if it's congestion control is based on delay.



- Initially, flow A takes 18 bandwidth, flow B takes 1.8, flow C takes 0.18
- A tax of 1%
- Flow A takes 18*0.99=17.84, flow B takes 1.8+0.18*10/11=1.97 flow C takes 0.18+0.18*1/11=0.197
- Repeat until flow B and flow C takes 10, flow A is bottlenecked at l_3
- Flow B is repeatedly taxed to redistribute bandwidth to flow C as it is the maximum flow at link l_2, until flow C saturates l_5
- Eventually, flow A is bottlenecked at l_3, flow B is bottlenecked at l_2, flow C is bottlenecked at l_5