软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2012,23(7):1924−1934 [doi: 10.3724/SP.J.1001.2012.04130]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种按需分配的多路径传输分组调度算法*

曹 宇[1,2,3+], 徐明伟[1,2]

[1](清华大学 计算机科学与技术系,北京 100084)

[2](清华信息科学与技术国家实验室(筹),北京 100084)

[3](解放军信息工程大学,河南 郑州 450002)

A Demand Based Packet Scheduling Algorithm for Multipath Transfer

CAO Yu[1,2,3+], XU Ming-Wei[1,2]

[1](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

[2](Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing 100084, China)

[3](Information Engineering University, Zhengzhou 450002, China)

+ Corresponding author: E-mail: caoyu08@csnet1.cs.tsinghua.edu.cn

Cao Y, Xu MW. A demand based packet scheduling algorithm for multipath transfer. *Journal of Software*, 2012,23(7):1924−1934 (in Chinese). http://www.jos.org.cn/1000-9825/4130.htm

Abstract: With the aid of multipath transport protocols, a multi-homed host can transfer data through multiple paths in parallel to improve goodput and robustness. However, the receiver has to deal with a large quantity of out-of-order packets due to the discrepancy of paths in terms of bandwidth, delay and packet losses. Theoretical analysis suggests that there are two approaches to reducing the memory overhead of caching out-of-order packets. One is to minimize the quantity of packets backlogged in outgoing queues of senders, and another is to decrease the packet sending rate. From the former, the study proposes a packet scheduling algorithm, named SOD (Scheduling On Demand), which assigns packets to each path according to the free window size. From the latter, a simple flow control method is proposed, which leverages the window feedback advertisement mechanism to limit the packet sending rate. Experimental results show that compared with existing algorithms, SOD suffers from the lowest memory overhead and obtains the highest goodput when receivers enable flow control. Additionally, SOD works steadily in the cases of diverse simulation scenarios.

Key words: multipath transfer; packet scheduling; out-of-order delivery; goodput

摘 要: 利用多路径传输协议,多宿主主机可以通过多条路径并行传输数据,从而有效提高系统的吞吐率和鲁棒性.但是由于不同路径在带宽、延迟和丢包率等方面存在差异,接收端必须缓存大量乱序到达的分组.数学分析表明,减少接收端的缓存开销有两条途径:一是最小化每条路径的发送队列中积压分组的数量,二是降低分组发送速率.由前者,提出依据每条路径的空闲发送窗口大小进行分组调度的算法 SOD(Scheduling On Demand);由后者,提出利用窗口通告机制限制分组发送速率的流控方法.模拟实验结果表明:与现有算法相比,SOD 的缓存开销最小;在接收端进行流控限制的情况下,SOD 的吞吐率最大,并且在不同实验场景中性能表现稳定.

* 基金项目: 国家自然科学基金(61073166, 61133015); 国家重点基础研究发展计划(973)(2009CB320502, 2012CB315803)
收稿时间: 2011-04-30; 修改时间: 2011-07-21; 定稿时间: 2011-10-08

关键词: 多路径传输;分组调度;乱序递交;吞吐率
中图法分类号: TP393 **文献标识码:** A

在当前的互联网中,具有多条并行接入通道的多宿主主机正逐渐增多[1],多宿主主机之间具有多条可达路径,这一特性为提升现有传输协议的性能提供了新的设计空间.多宿主主机主要来自两类应用场景:一是无线移动主机,包括各种智能终端;二是作为云计算基础设施的数据中心网络.无线网络技术的飞速发展,使得端系统接入网络的方法有多种选择.由于同一区域通常被多种网络覆盖(WiFi,3G 等),因此,移动主机可以并行利用多条接入链路获取更高的吞吐率和更强的鲁棒性.在数据中心网络中,主机之间有丰富的物理连接,采用多路径传输数据能够实现更灵活的负载均衡和更短的响应时延.

传统 TCP 协议只能使用端系统之间的一条可达路径传输数据,不能充分利用多宿主主机的多路径特性来提高传输性能.为此,学术界提出了多种多路径传输方案,比如 MPTCP[2],mTCP[3],pTCP[4]和 cTCP[5]等.然而,作为一种为应用层提供可靠字节流传输服务的协议,这些方案都存在的一个主要问题是分组的乱序到达和排序问题.在单路径传输的情况下,分组乱序到达接收端主要有两个原因:分组丢失和路由变化.依靠良好的确认重传机制,接收端可以灵活地选择缓存或者丢弃乱序分组.在多路径情况下,乱序问题更加显著,原因在于每条路径具有不同并且随机的延迟特性和拥塞状况.乱序分组会消耗接收端大量的缓存资源,并占用更多的处理器时间用于数据流的排序.传统 TCP 协议通过丢弃乱序分组来降低缓存和排序的开销,然而,这一策略并不适用于多路径传输方案,因为乱序分组来自不同的路径,接收端已经在各自的路径上向源端发出了确认消息,因此不能丢弃.对乱序分组延迟确认也不是一个合理的选择,因为这可能导致源端超时重传并降低发送窗口的大小,对协议性能产生不利影响.

现有的多路径传输方案大多采用轮转(round-robin)算法[2]调度分组,也就是当应用层发送数据时,分组被依次分配给每条路径进行传输.轮转算法是一种最简单、最直接的分组调度方法,但它没有考虑不同路径在带宽、延迟和丢包率等方面的差异,因而接收端必须开辟大量内存空间缓存乱序到达的分组.文献[2]对此进行了评估,建议把接收缓冲区的大小设为 $2\cdot\max\{RTT_i\}\cdot\sum BW_i$,其中,$RTT_i$ 是每条路径的往返传输时延,BW_i 表示路径的平均带宽.然而在某些场景中,这一开销却是不可接受的[2].比如有两条带宽为 100Mbps 和 1Mbps 的路径,往返传输时延分别是 10ms 和 500ms,那么根据前面的公式得出接收缓冲区应设为 12.5MB.文献[6]提出一种基于分组到达时间的负载均衡算法 ATLB(arrival-time matching load-balancing),它估算每条路径的发送队列中最后一个分组到达接收端的时延,把将要发送的分组分配给时延最小的路径,力求在多条路径上并行传输的分组尽可能地按序到达接收端.ATLB 在一定程度上能够减少乱序分组的数量,降低接收端缓存和排序的开销.然而,该调度算法的两个关键参数“往返传输时延”和“发送速率”存在估算误差.尤其是当路径的质量差异较大时,这种误差会显著影响算法的性能,本文的实验结果也印证了这一问题.

本文针对多路径传输中的分组乱序问题展开研究,主要贡献表现在如下方面:对接收端缓存乱序分组的数量进行了理论分析和推导.结果表明,可以采用两种方法减少这一开销:一是最小化每条路径的发送队列中积压分组的数量,二是降低分组发送速率.根据前者,本文提出发送端依据空闲发送窗口按需进行分组调度的算法,目标是在充分利用多条路径带宽资源以提高吞吐率的同时尽量减少乱序分组的数量;根据后者,本文提出接收端利用 TCP 窗口通告机制限制分组发送速率的流控方法,目标是把缓存开销控制在较小的水平.提高吞吐率与降低缓存开销是两个相互制约的目标,实验结果表明,与现有算法相比,按需调度算法能够在二者之间取得比较理想的平衡.

本文第 1 节简要介绍当前多路径传输的相关研究工作.第 2 节首先分析降低缓存开销的两条途径,然后提出按需调度算法,估算其缓存开销,最后给出流控方法.第 3 节采用输出队列长度和吞吐率两个性能指标对按需调度算法进行评价,并与轮转算法和 ATLB 算法进行对比.第 4 节总结全文.

1 多路径传输架构与相关研究工作

多宿主主机拥有多个并行网络接口,如果采用多路径传输方案,可以有效利用不同路径的带宽资源,提高系统的吞吐率和鲁棒性.当前,针对多路径传输协议的研究主要基于子流(subflow)和非子流两种架构,它们的区别在于是否在每条路径上按照传统 TCP 的工作模式传输分组.mTCP[3]和 cTCP[5]属于非子流架构的方案,它们用单一序列号空间标识分组,所有路径共享发送和接收缓冲区.因此,接收端从同一路径上收到的分组的序号通常是不连续的.为避免不必要的重传,mTCP 采用选择确认机制,并且挑选单一的反向路径发送确认消息;而 cTCP 则在发送端记录分组序号和路径的对应关系,通过启发式算法判断分组是否丢失.显然,这些方法增加了协议的复杂性,并且同一路径中不连续的分组序号可能会对某些防火墙等中间设备产生影响,不利于布署.MPTCP[2]和 pTCP[4]属于子流架构的方案,其特点是每条路径对应一条独立工作的子流,每条子流拥有独立的发送和接收缓冲区,分组除了被传统的序列号空间(连接级别序号)所标识,用于在接收端排序和重组以外,当它被调度到某条路径上传输时,会使用相应子流的序列号空间(子流级别序号)完成发送和确认,传输协议通过专门的映射机制实现分组在两个序号空间中的对应.从网络的角度看,子流的行为与传统 TCP 完全相同,所以不会对网络里的中间设备产生影响.

为保证与传统 TCP 协议兼容,多路径传输方案在建立连接的时候通过协商确定是否开启多路径传输功能,并且相互通告可用的接口地址.从端系统的角度看,一对源和目的地址代表一条可达路径,发送端根据调度算法并行从多条路径向接收端发送分组.pTCP 和 cTCP 根据路径的带宽按比例调度分组,前者用拥塞窗口和往返传输时延的比值估算每条路径的瞬时带宽[4];后者把丢包率也考虑进来,利用文献[7]中建立的模型估算每条路径的平均带宽[5].路径的带宽越高,分配给该路径的分组就越多,因而承载的流量就越大.在 mTCP 中,调度算法计算每条路径的待决数据量(即已发送并等待确认的数据)和拥塞窗口的比值,把新发送的分组分配给比值最小的路径[3].上述调度算法的目标都是实现多路径传输中的负载均衡,并没有关注分组乱序问题.然而在实际环境中,乱序分组的开销在某些情况下却是不可接受的[2],所以现有方案假设接收端拥有无限的缓存资源并不合理.

2 按需调度算法

本文提出的按需调度算法基于子流架构,并约定“路径”和“子流”表示相同的含义.

多路径传输协议与传统 TCP 一样,向应用层提供面向连接的、可靠的字节流传输服务,发送端对每个字节编号后送入网络,接收端根据编号按序把数据递交给应用层.为论述方便起见,本文约定以分组为单位进行编号.后文用到的符号参见表 1.

Table 1 Notations

表 1 符号说明

符号	含义	符号	含义
$\hat{w}_i$	子流 i 的平均发送窗口大小	τ_i	子流 i 的最后一个分组到达接收端的时延
$[x]^+$	如果 x 小于 0,则取 0;否则,取 x	Ω	子流集合
$\hat{rtt}_i$	子流 i 的平均往返传输时延	$flight_i$	子流 i 当前已发送并等待确认的分组数
sw_i	子流 i 当前的发送窗口大小	$highestAck_i$	子流 i 当前已确认的最高分组序号
$cwnd_i$	子流 i 当前的拥塞窗口大小	$nextTx_i$	子流 i 中下一个要发送的分组序号
win_i	子流 i 当前的流控窗口大小	$backlog_i$	子流 i 的发送队列中当前积压的分组数量
seq	分组在连接级别上的序号	$nextExp$	接收端在连接级别上期望收到的下一个分组的序号

2.1 分组调度算法

在给出按需调度算法之前,我们首先分析一下接收端的缓存大小与哪些因素相关.假设某一时刻发送端分别给子流 $i,j,\ldots,k$ 分配了 $n_i,n_j,\ldots,n_k$ 个分组,如图 1 所示,那么在子流 i 的最后一个分组到达接收端之前(这段时间设为 τ_i),通过 $j,\ldots,k$ 等其他子流到达接收端的分组只能存放在缓冲区中,其数量用 C_i 表示.在已知子流 s 的平均发送窗口 $\hat{w}_s$ 和往返传输时延 $\hat{rtt}_s$ 的条件下,可用它们的比值计算其平均发送速率,从而得到 C_i 的近似值为公式

(1).其中,子流的平均发送窗口由公式(2)定义.

$$\tilde{C}_i = \tau_i \cdot \sum_{s\in\Omega, s\neq i} \frac{\hat{w}_s}{r\hat{t}t_s} \tag{1}$$

$$\hat{w}_s = \lim_{T\to+\infty} \frac{\int_0^T sw_s(t)\mathrm{d}t}{T}, s\in\Omega \tag{2}$$

为进一步求得τ_i,我们把子流发送分组的行为抽象成“轮”模型[7],即每个往返传输时延称为一轮(round).因为子流 i 每轮平均发送 $\hat{w}_i$ 个分组,所以子流 i 在发送最后一轮分组之前,经过的轮数可近似表示为 $[n_i-\hat{w}_i]^+/\hat{w}_i$,而最后一轮分组只需一半的往返传输时延即可到达接收端,因此得到τ_i的近似值为公式(3).

$$\tilde{\tau}_i = \left(\frac{[n_i-\hat{w}_i]^+}{\hat{w}_i} + \frac{1}{2}\right)\cdot r\hat{t}t_i \tag{3}$$

从公式(1)可以看出,最小化 C_i 的值有两条途径:

一是减少 $\hat{w}_s$,即降低每条子流的分组发送速率,但这也意味着降低吞吐率;

二是减少τ_i.由公式(3)可知,当 $n_i \leqslant \hat{w}_i$ 时, $\tilde{\tau}_i$ 达到最小值 $r\tilde{t}t_i/2$.换句话说,只要每次调度给子流的分组不积压在发送队列中,那么就能够使接收端的缓存开销最小.

根据这一结论,本文提出按需分配的分组调度算法 SOD(Scheduling On Demand).

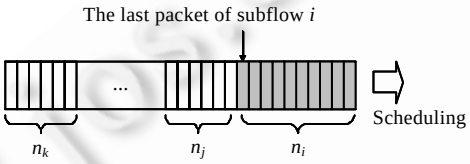


Fig.1 A plan of packet scheduling

图 1 分组调度计划

SOD 算法采用输入队列(input queue)和发送队列(outgoing queue)两级缓冲结构,如图 2 所示.来自应用层的数据流被划分成分组放入输入队列,当某一子流的发送队列中积压分组的数量小于空闲发送窗口时,调度算法从输入队列依次分配分组到该子流的发送队列,直到空闲的发送窗口被填满.当子流超时或者路径失效时,其发送队列中的分组重新按序插入到输入队列,等待被调度到其他子流上传送.算法用 Python 语言表述于表 2.

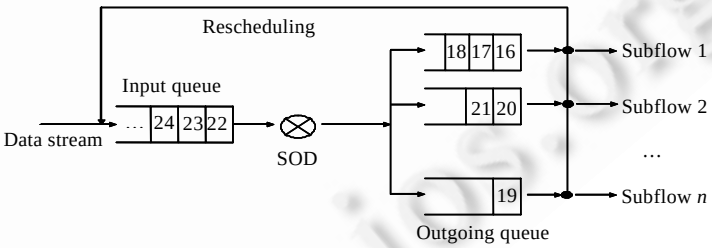


Fig.2 The framework of packet scheduling at senders

图 2 发送端分组调度架构

Table 2 SOD algorithm

表 2 SOD 算法

Algorithm 1. SOD.

```
while True:
    for s in Ω:
        if s is broken down or has timed out:
            Move all the packets in the outgoing queue of s to the input queue
            continue
        sw_s=min(cwnd_s,win_s)
        flight_s=nextTx_s−highestAck_s
        if flight_s>=sw_s or backlog_s>=(sw_s−flight_s):
            continue
        Move (sw_s−flight_s−backlog_s) packets from the input queue to the outgoing queue of s
```

2.2 接收端的缓存开销

本文用乱序分组的数量作为接收端缓存开销的度量.从公式(1)不难看出,慢速子流是影响缓存开销的主要因素.比如有 s 和 p 两条子流,往返传输时延分别是 20ms 和 200ms,发送窗口始终为 1,那么在子流 p 的分组到达接收端之前,来自子流 s 的分组必须全部暂存在缓冲区中,数量为(200/2)×(1/20)=5,而子流 s 的分组则不会对子流 p 产生影响.

推广到一般情况,因为在路径不发生拥塞时发送队列中没有积压的分组,即满足条件 $n_i \leqslant \hat{w}_i$, 所以由公式(3)可知, $\tilde{\tau}_i = r\hat{t}t_i/2$, 再由公式(1)可得 SOD 算法在稳定状态下的缓存开销为公式(4).

$$\tilde{C}_{stable} = \max_{i\in\Omega}\left\{\frac{r\hat{t}t_i}{2}\cdot\sum_{s\in\Omega,s\neq i}\frac{\hat{w}_s}{r\hat{t}t_s}\right\} \tag{4}$$

为了进一步确定缓存开销的上限,我们考虑子流超时或者路径失效的情况.根据 SOD 算法,如果子流 i 发生超时,其发送队列中的分组会转移到输入队列重新进行调度.为简化推导,假定在分组重新调度期间不会有新的路径失效.子流 i 检测超时需花费 RTO(retransmission time-out)的时间,它近似等于两倍的往返传输时延,这段时间接收端缓存的分组数为 $2\cdot r\hat{t}t_i\cdot\sum_{s\in\Omega,s\neq i}\hat{w}_s/r\hat{t}t_s$. 因为在输入队列中标记为重新调度的分组具有最小的序号,所以会被优先分配给可用的子流.这意味着在这些分组发送成功之前,接收端缓存的分组数量最多增加 $\sum_{p\in\Omega,p\neq i}\hat{w}_p$.

综上,得到 SOD 算法缓存开销的上限为公式(5).

$$\tilde{C}_{upper} = \max_{i\in\Omega}\left\{2\cdot r\hat{t}t_i\cdot\sum_{s\in\Omega,s\neq i}\frac{\hat{w}_s}{r\hat{t}t_s}+\sum_{p\in\Omega,p\neq i}\hat{w}_p\right\} \tag{5}$$

2.3 流控方法

在基于子流的多路径传输架构中,每个分组被两个序列号空间标识,即“连接级别序号”和“子流级别序号”.所以,分组在接收端必须首先进入相应子流的接收队列(ingoing queue),利用子流级别序号向源端发送确认消息,然后才能提交给输出队列(output queue),在那里通过连接级别序号进行排序和重组,最后递交给应用层.因此,接收端的队列结构如图 3 所示.

子流接收队列的容量通常是固定的,但输出队列必须足够容纳所有连接级别乱序到达的分组.所以,接收端的缓存开销主要是指输出队列的长度.前面的讨论隐含假设了输出队列具有无限容量,虽然 SOD 算法在稳定状态下的缓存开销较小,但当子流发生超时或者路径失效时,由公式(5)可知,接收端仍需缓存大量分组.为了进一步降低开销,根据公式(1)的分析结果,可采用限制分组发送速率的方法.

对每条子流而言,因为接收队列的容量是固定的,所以最直观的速率限制方法是把乱序分组保留在相应子流的接收队列中,通过传统 TCP 的窗口通告机制降低该子流的发送速率,从而减少输出队列的长度.

然而,我们的这个初步想法却面临“队头阻塞”问题.比如有 s 和 p 两条子流,由于子流 s 发生拥塞导致从子流 p 上到达接收端的分组大量缓存在接收队列中,无法向输出队列递交.当子流 p 的接收队列耗尽时,传输被迫停

滞,进入零窗口探测期.如果此时子流 s 失效,则其发送队列中的分组只能转移到子流 p 进行传输,从而导致阻塞.

为解决上述问题,我们采用如下方法:当子流的接收队列耗尽时,即使分组是乱序的,仍然向输出队列提交δ个分组,以避免传输发生停滞.模拟实验表明,δ=2 即可取得良好效果.流控方法用 Python 语言表述于表 3.

需要说明的是,当某一子流向输出队列提交分组后,必须检查其他子流是否也有可以提交的分组.这一过程反复执行,直到所有子流的接收队列中都不存在下一个期望接收的分组为止.为了表述简洁,上述处理过程仅用第 11 行语句来描述.

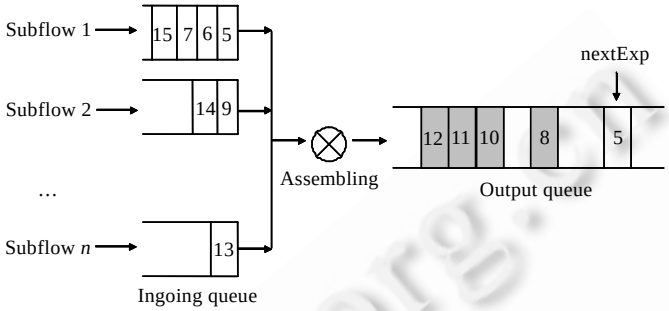


Fig.3 The framework of packet assembling at receivers

图 3 接收端分组排序架构

Table 3 Flow control method

表 3 流控方法

	Algorithm 2. Flow control.
1	Subflow *s* receives a packet
2	*seq*=the sequence number of the received packet at connection level
3	if *seq*>*nextExp*:
4	if the ingoing queue of *s* is exhausted:
5	Move δ packets from the ingoing queue of *s* to the output queue
6	else:
7	Keep the received packet staying at the ingoing queue of *s*
8	else if *seq*==*nextExp*:
9	Move the received packet from the ingoing queue of *s* to the output queue
10	*nextExp*=*nextExp*+1
11	Check other ingoing queues and move packets to output queue if needed
12	else:
13	Discard the received packet
14	Advertise the free size of the ingoing queue of *s* to the sender

3 性能评价

尽管多路径传输协议目前还没有严格的协议标准,但我们利用 RFC 6182[2]及相关草案[8,9]中描述的框架和机制,在 ns-3 模拟平台[10](版本 3.9)上实现了多路径传输协议的基本功能以及 Round-Robin(RR),ATLB 和 SOD 这 3 种分组调度算法.

性能评价的指标有两个,分别是应用层的实际吞吐率(goodput)和用于乱序分组排序的输出队列的长度(memory overhead,即缓存开销).

网络拓扑如图 4 所示,其中,S_i 和 D_i(i=1,2,…,n)分别表示发送端和接收端的接口地址.发送端首先用 S_1 连接 D_1,建立第 1 条子流.在这一过程中,接收端通过该子流的 TCP 选项域通告可用的其他地址(D_2,…,D_n),从而使发送端有足够的信息建立多条子流.虽然理论上子流数量的上限是发送端和接收端地址数的乘积,但实际的子流数应该与端系统之间不相交路径的数量保持一致.因此,我们在协议实现中规定:当入站分组的目的地址与接收网卡的接口地址不相同时,丢弃该分组.所以对于图 4 所示的网络拓扑而言,子流数量是 n.

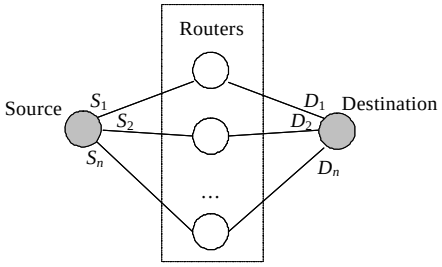


Fig.4 Network topology for simulation

图 4 实验的网络拓扑

与实验相关的参数如下:每条子流的发送队列容量是 32KB,接收队列容量是 64KB.每个路由器结点以随机的方式向接收端发送强度为 0.8Mbps 的 UDP 数据流,用于模拟背景流量.端到端持续传输时间是 60s.若不作特别说明,所有链路的带宽都是 2Mbps,丢包率为 0.001.

表 4 列出了实验的 6 种场景.以场景 1 为例,场景名称“(280, 40)”表示端系统之间只有两条可达路径,其最小往返传输时延分别是 280ms 和 40ms.也就是说,在第 1 条路径中,每条链路的延迟设定为 70ms,第 2 条路径中设定为 10ms.

需要说明的是,由于存在背景流量,实际的往返传输时延是大于设定值的.此外,场景 5 的名称表示在第 3 条路径中每条链路的延迟是 80ms,带宽为 1.5Mbps,丢包率设为 0.01,它模拟了一条网络状况较差的路径.

Table 4 Simulation of scenarios

表 4 实验的场景

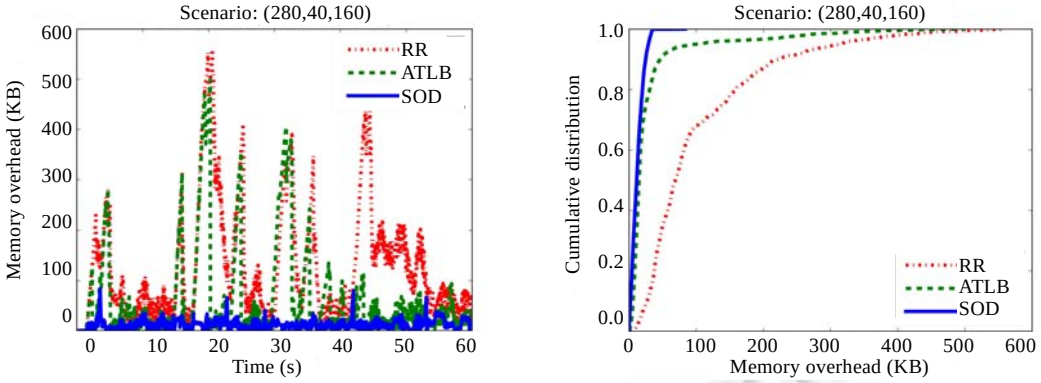
编号	场景名称	说明
1	(280,40)	基本场景
2	(280,40,280)	在场景 1 的基础上增加一条慢速路径
3	(280,40,160)	在场景 1 的基础上增加一条中速路径
4	(280,40,40)	在场景 1 的基础上增加一条快速路径
5	(280,40,320 1.5 0.01)	在场景 1 的基础上增加一条质量较差的路径
6	(280,40,320,20)	在场景 1 的基础上增加两条差别较大的路径

3.1 接收端输出队列的长度

因为每条子流的接收队列容量是固定的,所以我们用输出队列的长度作为缓存开销的度量.在没有流控限制的条件下,图 5(a)显示了在端到端有 3 条路径的场景中 3 种调度算法的缓存开销.从图中不难看出,SOD 算法的缓存开销变化幅度较小,且总体上低于 RR 和 ATLB 算法.

为了更清晰地进行比较,图 5(b)给出了缓存开销的累积分布曲线,而图 6 则展示了另外两种场景的实验结果.虽然在路径数量较少时 ATLB 和 SOD 性能接近,但前者的分布曲线具有较长的拖尾,并且随着路径数量的增加,其性能明显降低.原因在于,ATLB 算法用于分组调度的两个关键参数“往返传输时延”和“子流的瞬时发送速率”极易受到网络状态随机波动的影响,这对于时延较短的路径尤为明显.当接收端进行流控限制后,缓存开销会大大降低.

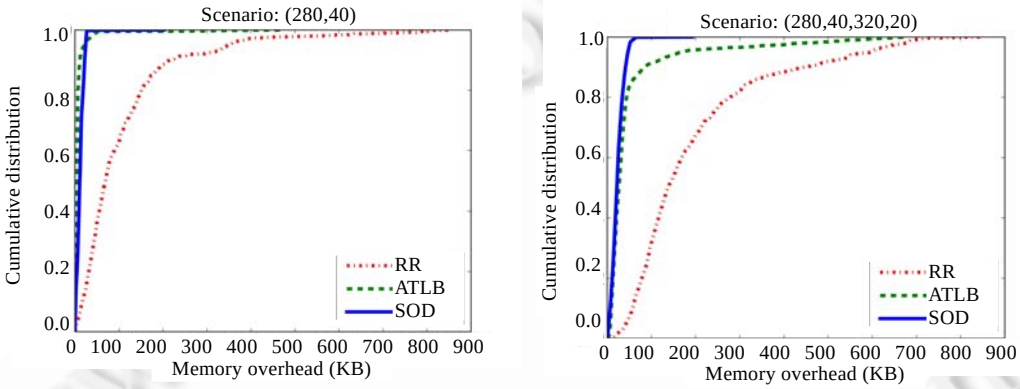
图 7 显示了在 3 条路径的场景中,流控前后输出队列的最大长度.这一结论具有普遍性,限于篇幅,这里不再列举其他场景中的实验结果.



(a) Fluctuations (b) Cumulative distribution

Fig.5 Length of the output queue, disabling flow control

图 5 输出队列的长度,无流控



(a) 2 subflows (b) 4 subflows

Fig.6 Cumulative distribution of the output queue length, disabling flow control

图 6 输出队列长度的累积分布曲线,无流控

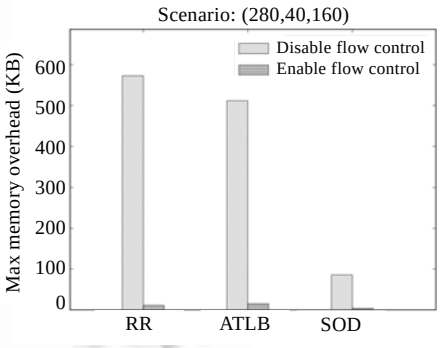


Fig.7 Maximum memory overhead

图 7 输出队列的最大长度

3.2 吞吐率

流控方法对吞吐率的影响主要有 3 个方面.

1) 当没有流控限制时,3 种调度算法的吞吐率大致相当,如图 8 中的 disable flow control 所示,但缓存开销有

较大差别(已在上节有所分析).

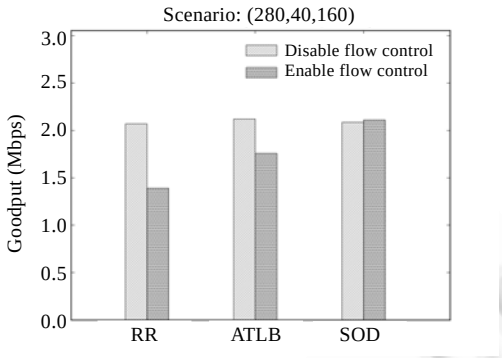
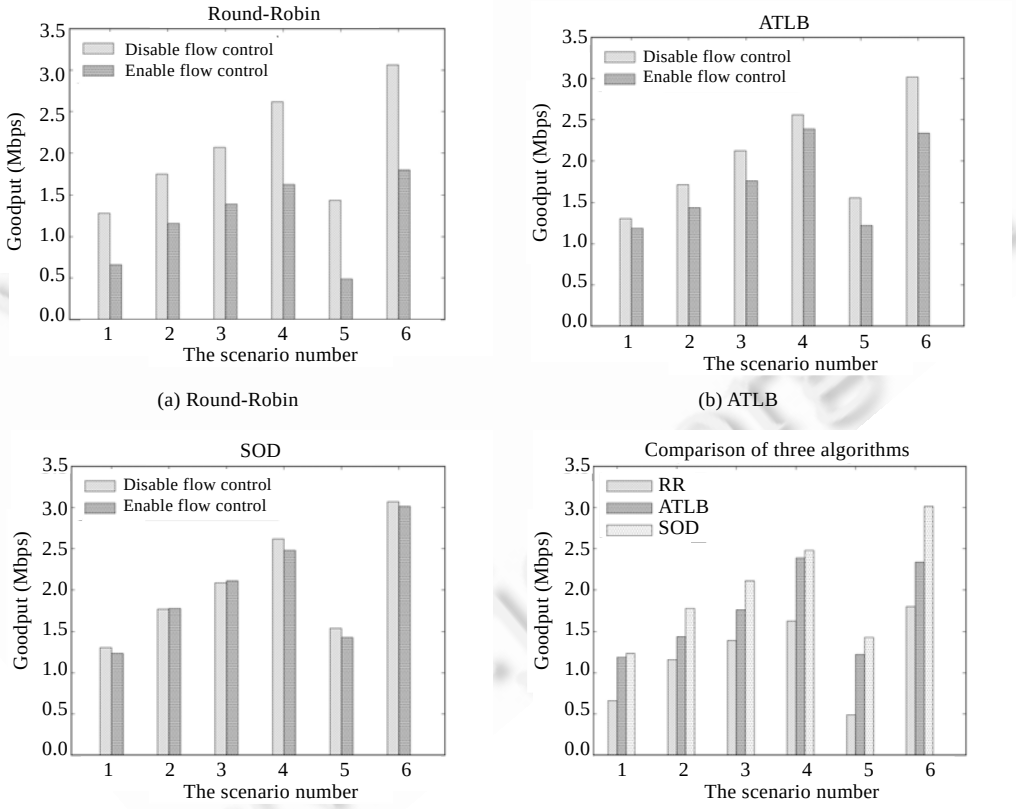


Fig.8 Goodput, 3 subflows

图 8 吞吐率,3 条子流

2) 一旦接收端进行流控限制(δ=2),在降低缓存开销(图 7 中的 enable flow control)的同时,吞吐率也会随之下降.在流控的条件下,为全面客观地比较 3 种调度算法的性能,我们在表 4 列出的 6 种场景中进行了实验,结果显示在图 9 中.总体来看:RR 的吞吐率下降幅度最大;ATLB 次之;而 SOD 最小,甚至在某些场景中几乎不发生变化.图 9(d)是在流控的条件下对 3 种算法进行的综合比较,显然,SOD 的吞吐率在所有场景中都是最高的.



(a) Round-Robin (b) ATLB

(c) SOD (d) 3 algorithms, enabling flow control

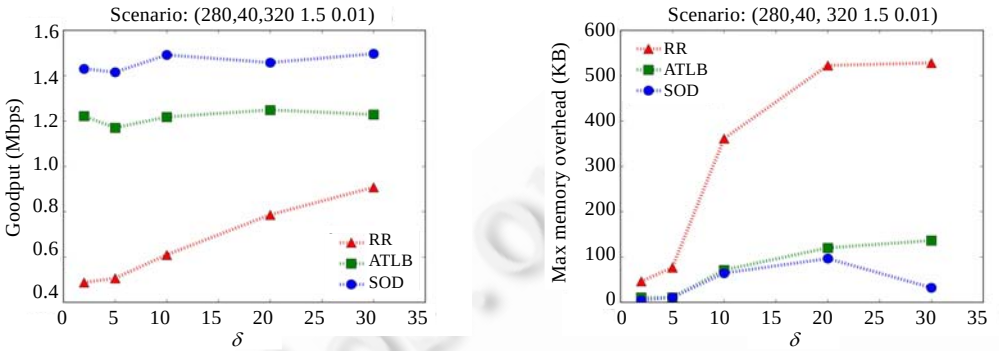
Fig.9 Goodput of 3 algorithms in different scenarios

图 9 3 种调度算法在不同场景中的吞吐率

3) 流控方法中,δ参数对吞吐率的影响将在下一节加以论述.

3.3 δ参数

在流控方法中引入参数δ的目的是为了解决“队头阻塞”问题,所以直观来看,当端系统之间的多条路径质量差别较大时,接收队列中缓存的乱序分组会比较多,调度算法的性能对δ的取值也较为敏感.因此,我们选择场景“(280,40,320 1.5 0.01)”进行实验.从图 10 可以看出,随着参数δ的增加,RR 算法的吞吐率直线上升,同时,其最大缓存开销也迅速增长.这一现象从另一个角度印证了 RR 算法的内存资源消耗较大.ATLB 和 SOD 的最大缓存开销也会缓慢增长,然而,其吞吐率却只有轻微波动.实际上,这些结论对于其他场景同样也成立.因此,对 SOD 算法而言,δ取 2 是一个比较合理的选择.



(a) Goodput (b) Maximum memory overhead

Fig.10 Performance of SOD with the different δ

图 10 参数δ对性能的影响

3.4 路径质量对吞吐率的影响

从前面的实验可以看出:在接收端没有流控限制的前提下,3 种调度算法能够获得大致相同的吞吐率;并且随着路径数量的增加,吞吐率也不断增大.然而,当接收端进行流控限制后,上述结论却不一定成立.因为慢速子流会“阻塞”快速子流,导致系统整体性能下降,比较图 9(a)的场景 1 和场景 5 以及图 9(b)的场景 4 和场景 6 均会观察到这一现象.所以在多路径传输中,路径的质量比数量更重要.此外,从图 9 中得到的另一个结论是,RR 和 ATLB 算法对路径质量的差异性比较敏感,而 SOD 算法却相对稳定.

4 总 结

在多路径传输中,分组调度是一个关键问题,现有方案通常采用轮转算法调度分组,并且假设接收端拥有无限的缓存资源,然而这并不能满足实际的应用需求.ATLB 算法虽然能够有效减少乱序分组的数量,但其通过测量得到的调度参数容易受到网络状态随机波动的影响,从而削弱了实际的传输性能.本文提出的 SOD 算法具有简洁、高效和稳定的特点,与前两种方法相比,它的缓存开销最小,并且对流控限制和路径质量差异性具有良好的适应能力,在各种场景中性能表现也比较稳定.提高多路径传输的吞吐率与降低接收端的缓存开销是两个相互制约的目标,本文提出的流控机制只是均衡二者的一种最直观的方法,对此仍需进行深入研究和讨论.

References:

[1] Meyer D, Zhang LX, Fall K. Report from the IAB workshop on routing and addressing. RFC4984, 2007. http://www.ietf.org/rfc/rfc4984.txt

[2] Ford A, Raiciu C, Handley M, Barre S, Iyengar J. Architectural guidelines for multipath TCP development. RFC6182, 2011. http://tools.ietf.org/html/rfc6182

[3] Zhang M, Lai JW, Arvind K, Larry P, Randolph W. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In: Proc. of the USENIX 2004 Annual Technical Conf. Boston: USENIX Association, 2004. 99−112. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.111.3109

[4] Hsieh HY, Raghupathy S. pTCP: An end-to-end transport layer protocol for striped connections. In: Proc. of the 10th IEEE ICNP. Paris: IEEE Computer Society, 2002. 24−33. [doi: 10.1109/ICNP.2002.1181383]

[5] Dong Y, Wang DD, Niki P, Wang J. Multi-Path load balancing in transport layer. In: Proc. of the 3rd EuroNGI Conf. Trondheim: IEEE Computer Society, 2007. 135−142. [doi: 10.1109/NGI.2007.371208]

[6] Yohei H, Ichiro Y, Takayuki H, Hideyuki S, Tutomu M. Improved data distribution for multipath TCP communication. In: Proc. of the IEEE GLOBECOM 2005. St. Louis: IEEE Computer Society, 2005. 271−275. [doi: 10.1109/GLOCOM.2005.1577632]

[7] Jitendra P, Victor F, Don T, Jim K. Modeling TCP throughput: A simple model and its empirical validation. In: Proc. of the ACM SIGCOMM'98. Vancouver: ACM Press, 1998. 303−314. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.143.9137 [doi: 10.1145/285237.285291]

[8] Barre S, Paasch C, Bonaventure O. Multipath TCP—Guidelines for implementers. IETF Internet Draft, 2011. http://tools.ietf.org/html/draft-barre-mptcp-impl-00

[9] Ford A, Raiciu C, Handley M. TCP extensions for multipath operation with multiple addresses. IETF Internet Draft, 2011. http://tools.ietf.org/html/draft-ford-mptcp-multiaddressed-03

[10] The ns-3 network simulator. http://www.nsnam.org/

曹宇(1981－),男,甘肃天水人,博士生,讲师,主要研究领域为互联网路由.

徐明伟(1971－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为未来互联网体系结构,大规模路由,网络虚拟化.