SSRC：时延敏感流的数据源端速率控制算法

杨洋[1]，曹敏[1]，杨家海[2,3]，车嵘[1]，刘伟[1]
（1. 国防科技大学信息通信学院，陕西 西安 710106；2. 清华大学网络科学与网络空间研究院，北京 100084；
3. 清华信息科学与技术国家实验室（筹），北京 100084）

摘 要：当前的研究工作针对如何保证时延敏感流的传输时间进行了大量研究，但普遍存在时效性不够强的问题。基于 SDN/OpenFlow 架构，提出了数据源端控制算法 SSRC。该算法依据网络的全局视图，快速定位拥塞可能发生的节点，并及时对目标流的源端速率进行调节，可以缩短算法的响应时间。实验结果表明，与 DCTCP 等算法相比，所提算法的流完成时间平均缩短了 75%，且能够保证时延敏感流的传输时间，很好地解决 Incast 问题。

关键词：时延敏感；长流；软件定义网络；缓存溢出
中图分类号：TP393
文献标识码：A
doi: 10.11959/j.issn.1000−436x.2019070

SSRC: source rate control algorithm for delay-sensitive flow in data center network

YANG Yang[1], CAO Min[1], YANG Jiahai[2,3], CHE Rong[1], LIU Wei[1]
1. School of Information and Communication, National University of Defence Technology, Xi'an 710106, China
2. Institute for the Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China
3. Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing 100084, China

Abstract: Previous work had studied how to ensure the transmission time of delay-sensitive flows, but falled short in its effectiveness for a given period. Motivated by that and based on SDN/OpenFlow framework, a SDN-based source rate control (SSRC) algorithm was proposed. With a global view of network, SSRC can quickly locate the potential congestion node, adjust the transmission rate of source and reduce the response time of SSRC. The experiment results show that compared with DCTCP and other algorithms, SSRC can shorten the completion time of flows by 75% in average, which can ensure the transmission time of delay-sensitive and solve the problem of Incast well.

Key words: delay-sensitive, long flow, software-defined network, buffer overflow

1 引言

当前，数据中心 95%以上的数据流都是 TCP 流[1]，随着业务种类急剧增长，流量的多样性越来越突出。这些流量可以分为 2 类：一类是吞吐量敏感型的流量，例如，大量云计算业务由于虚拟机迁移、数据备份等操作产生的长流；另一类是时延敏感型的流量，例如，Web 请求（包括搜索流量）业务、基于 MapReduce[2]的并行计算业务以及社交网络等在线业务产生的短流。数据中心为业务提供高可用聚合带宽保证的同时，网络拥塞也频繁发生，同时网络中出现的 Incast 现象逐渐成为数据中心区别于传统互联网的标志性事件。由于当前数据中心产生的短流类型业务大部分是有流传输截止时间要求的业

收稿日期：2018–10–09；**修回日期**：2019–02–13
基金项目：国家重点研发计划基金资助项目（No.2016YFB0801302）；国防科技大学计划科研基金资助项目（No.ZK18-03-59）
Foundation Items: The National Key Research and Development Program of China (No.2016YFB0801302), The Research Program of National University of Defence Technology (No.ZK18-03-59)

务，一旦传输路径中出现链路拥塞，就会导致传输超时并严重影响用户体验。由于 TCP 长流的“贪婪性”会导致交换机缓存中数据分组排队队列增长，直到缓存溢出产生分组丢失。如果链路中长短流共存，会导致 2 种情况发生：一种是短流分组丢失导致 Incast 出现；另一种是虽然短流分组不丢失，但由于在缓存中排队的短流数据分组转发优先权低于长流数据分组，导致短流数据排队时间超出流传输截止时间。由于数据中心短流具有突发、不可预测特性，网络中任何一条传输链路都有可能存在因链路拥塞而产生的长短流碰撞，导致出现短流分组丢失的可能性。因此，为保证数据中心时延敏感流的传输时间，有必要在数据源端对引起交换机缓存队列长度超过阈值的长流进行速率调节，以避免链路中长短流碰撞而导致短流分组丢失。

当前数据中心产生短流类型的业务大部分是有流传输截止时间要求的业务，例如，Web 请求类业务、基于 MapReduce 并行计算业务等，一旦传输路径中出现链路拥塞，就会导致传输超时并严重影响用户体验。另外，由于网络中存在大量采用分割/汇聚（partition/aggregate）技术的业务，例如，MapReduce 以及搜索请求业务等，这种“多对一”的通信方式带来的 TCP Incast 问题也越来越突出[3]。由于 Incast 问题通常是由承载汇聚信息的短流分组丢失引起的，因此，Incast 问题的解决可以转化为针对时延敏感的短流进行避免传输链路拥塞的研究[4-6]。当前针对数据中心时延敏感流量进行数据源端速率控制的研究工作可以分为以下 2 类。

1) 终端设备解决算法

终端设备的解决算法是指在数据发送端通过调整发送窗口值的大小来控制发送速率的算法。例如，TCP Newreno 是对 TCP 的改进版本，通过引入快速恢复机制避免了快速重传之后马上进入慢启动阶段而导致发送窗口减小过大的问题，是当前网络通信普遍采用的协议。文献[7]通过将最小超时重传时间 RTO_{min}（retransmission time out）减小到微秒级，缓解由于产生 Incast 问题而造成的吞吐量下降。另一类算法是设计改进的 TCP，例如，D2TCP[8]通过提前获取数据流传输的截止时间，并根据流截止时间作为调整拥塞窗口的惩罚因子，计算拥塞窗口大小，达到通过控制数据流传输速率来消除拥塞的目的。ICTCP[9]以接收端实际测量吞吐量与期望吞吐量之间的差值比率是否超过某个阈值，作为调整接收窗口大小的依据，并将调整的窗口信息以 ACK 信号反馈给发送端，达到控制数据流速率的目的，从而避免链路拥塞导致的短流分组丢失。文献[10]提出 MMPTCP（maximum multi-path TCP），在终端设置数据流传输阈值，数据流传输初始阶段是基于分组粒度的多路径负载均衡，使时延敏感型的数据流受益；当传输数据量超过阈值时，进入第二阶段，即由基于数据分组粒度的多路径负载均衡方式切换到 MPTCP（multi-path TCP）[11]模式，有效地保证长流吞吐量。文献[3]则提出了应用层的解决算法，以一定的随机概率刻意延迟服务器对请求的响应，从而在某个时间段减少同时参与请求响应的服务器数量，解决 Incast 的同步屏障问题。

2) 交换设备支持的解决算法

交换设备支持的解决算法是指由交换机（或路由器）与终端主机共同解决拥塞避免的算法。例如，DCTCP（data center TCP）[1]基于 ECN（explicit congestion notification）[12]的功能，通过设置交换机缓存队列长度阈值，对超过阈值的数据分组进行标记，数据源端则根据反馈的拥塞程度信息按照一定的衰减因子动态调整发送窗口，被标记的分组数量越多、衰减因子越大，发送窗口就越小，从而始终保证交换机队列长度低于某个阈值，防止由于缓存溢出而丢弃数据分组。D3[13]则采用带宽资源预留的方式，提前为数据流分配所需传输带宽。数据源端首先获取数据流截止时间以及流大小信息，在每一轮 RTT（round trip time）周期内发送相关传输数据的带宽需求信息，交换机收到该信息后计算出预留带宽并将预留带宽值写入数据分组头。数据分组传输路径上的所有交换机将执行相同的操作，从而保证数据流在截止时间内传输完毕。另外还有通过交换机向数据源端发送“暂停”帧实现流控的 EFC（ethernet flow control）[14]机制以及基于 IEEE 802.1Qau 以太网标准提出的链路层拥塞控制算法 QCN（quantized congestion notification）[15]。

尽管当前的研究算法对于避免数据中心出现拥塞链路而导致短流分组丢失，以及针对 Incast 问题的解决都起到了积极的作用，然而在实际部署的可扩展性、实施开销以及时效性方面都存在不足。

其中，终端设备的解决算法对于 TCP 的改进方法需要接入终端设备修改协议栈，实施难度高，可扩展性不强。例如，D2TCP 、ICTCP、MMPTCP 都需要在终端进行阈值的设定以及相应的判断、计算操作，D2TCP 还需要提前获取数据流截止时间等信息；对 TCP 参数进行调整实施难度虽然低，但是需要其他机制配合，否则效果不理想。例如，减小 RTO_{min} 虽然可以提升吞吐量，但也会导致欺骗性重传。交换设备参与解决的算法需要设备的硬件支持，例如，DCTCP 需要在交换机缓冲区的队列长度达到阈值时对数据分组进行标记；D3 则需要交换机根据数据源端的带宽需求信息计算出预留带宽，并要求传输路径上的所有交换机对这条数据流执行相同的操作；EFC 和 QCN 都需要特定的交换机支持。交换设备参与的解决算法对设备硬件要求高，部署开销较大，尤其对于数据中心大多使用低成本的商用交换机的情况，更不适合大规模的部署。另外，基于 TCP 连接的反馈回路调节源端速率的算法还存在时效性不强的问题，例如，当反馈回路出现链路拥塞或者发生故障时，将影响算法执行的效率。

综上所述，本文基于 SDN/OpenFlow 的架构，提出了数据源端控制算法 SSRC（SDN-based source rate control）。SSRC 依据网络的全局视图，能够快速定位可能发生拥塞的节点，并及时对目标流的源端速率进行调节，缩短算法的响应时间。本文的主要贡献如下。

1) 利用 SDN 的架构设计能够对链路中出现的长流以及交换机缓存的队列长度进行监测，快速定位拥塞可能出现的位置，并确认需要进行源端速率调节的目标长流。

2) 控制器利用目标长流建立反馈回路，修改携带接收窗口大小的 TCP_ACK 数据分组，并直接将该数据分组推送到连接数据源端的接入层交换机，极大地缩短了算法机制的响应时间，提高了算法的时效性。

3) 通过将 NS3 网络仿真工具与 FloodLight 外部控制器相结合，形成基于 SDN 架构的网络仿真平台，仿真实验结果证明 SSRC 能够保证时延敏感流的传输时间，同时能够很好地解决 Incast 问题。

2 算法设计关键问题分析

TCP 长流的“贪婪性”会导致交换机缓存中数据分组排队队列增长，直到缓存溢出产生分组丢失。如果链路中长短流共存，会导致 2 种情况发生：一种是短流分组丢失导致 Incast 出现；另一种是虽然短流分组不丢失，但由于缓存中排队的短流数据分组转发优先权低于长流的数据分组，导致排队时间超出流传输截止时间。当前在集群化存储的系统内，客户端的应用请求都以服务请求数据单元（SRU, service request unit）方式分别存储在服务器上，只有当客户端收到所有服务器的 SRU 后，才能继续下一个服务请求。然而在链路出现拥塞造成排队队列处理时延增大甚至导致短流分组丢失严重时，会使完成一个 TCP 应用请求至少经历 200 ms 的超时[16]。对于时延敏感的业务，将严重影响用户体验；对于 MapReduce 之类的并行计算业务，将严重浪费计算资源。所以，时延敏感的短流受链路拥塞影响最大，其根本原因是交换机缓存队列的积压导致链路中出现的长流与时延敏感的短流出现碰撞，造成短流分组丢失。

D2TCP 和 D3 都是针对时延敏感的短流提出的解决算法，但是两者都需要提前获取数据流的截止时间，然而实际中这样的信息可能无法提前获取。ICTCP 主要解决 Incast 问题，但只是针对最后一跳的链路提出的解决算法，没有考虑承载 SRU 的短流分组丢失发生在中间交换节点的情况，并且通过接收端窗口大小来控制源端发送速率需要通过 ACK 将窗口信息反馈回发送端，如果反馈回路拥塞或者发生故障则严重影响算法的执行效率。DCTCP 通过设定交换机队列长度阈值的方式并基于 ECN 将拥塞程度信息反馈回发送端，这样的算法除了对交换设备要求高之外，也存在反馈回路影响算法执行效率的问题。

综上所述，本文提出基于 SDN/OpenFlow 框架的解决算法，能够克服现有算法存在的不足。由于 SDN 具有集中化管控的优势，控制器拥有全局的网络资源视图，因此更容易提前发现可能出现拥塞的节点，通过控制器下发策略避免拥塞；其次，当发现可能的拥塞节点后，控制器能快速进行响应，相对于传统网络中接收端通过反馈回路进行数据源端速率调节的算法，能够极大地缩短响应时间，提高算法的时效性。本文提出的基于 SDN/OpenFlow 的数据源端速率控制算法需要解决 2 个关键问题：一是设计算法的触发机制，二是需要获取数据源端优化后的目标发送速率。

3　算法设计

3.1　算法触发机制

由于交换机缓存队列积压而出现长短流碰撞是导致短流分组丢失的根本原因。同时，必须注意到分组丢失发生的位置除了包含最后一跳网络节点设备外，网络中间设备都有可能由于交换设备的缓存溢出而导致分组丢失。因此，算法设计的目标是能够通过全局网络视图，提前发现有可能出现拥塞的网络节点，通过触发算法避免链路由于 TCP 长流的“贪婪性”造成交换机排队队列长度增加从而导致短流分组丢失。因此，算法的触发机制由 2 个关键条件决定：一是链路中出现长流，二是出现长流的交换设备中的队列长度超过阈值。当 2 个条件同时满足时算法被触发，这样设计的目的是，针对链路中容易引起短流分组丢失的目标长流，快速进行拥塞避免策略响应，增强算法执行的时效性。

1) 长流的发现

当前 OpenFlow 的版本都支持 2 种方式统计数据流信息[17]：一种是基于控制器发送 Read_State 消息，对交换机状态信息采用轮询的方式统计；另一种由交换机发送异步消息对控制器进行数据流信息的推送。由于交换机推送数据流信息的方式是当该流传输结束或者流表删除时向控制器推送消息，并不适合对长流的探测，因此本文采用的方法是控制器以周期轮询的方式获取交换机相关数据流信息，并以此发现长流。采用轮询的方式探测长流属于 OpenFlow 自带的原生测量，不会产生额外的探测开销，但需要注意的是轮询周期不能设置过小，否则会加重控制器与交换机之间的通信负担。通过对比实验，在保证能够探测到目标长流的前提下，将控制器轮询周期设置为 5 s。

2) 队列长度阈值

设置交换机缓存队列长度阈值 K_t，如式(1)所示。

$$K_t > \frac{C \times \mathrm{RTT}}{7} \tag{1}$$

其中，C 代表瓶颈链路带宽，单位为 packet/s，即每秒传输数据分组的数量；RTT 代表往返时延，单位为 s。在数据中心实际环境中，考虑到流量的突发特性，K_t往往不能取下限值，通常当链路带宽为 1 Gbit/s 时，设置 K_t= 20；当链路带宽为 10 Gbit/s 时，设置 K_t=65。

3.2　算法优化问题

当算法触发时，控制器需要计算出合理的数据源端发送速率，而发送速率值是由数据源端的发送窗口大小决定的。以往的研究工作中，对于数据源端发送窗口大小的反馈调节机制大部分都是基于一个前提条件，即 N 个数据源端发送的响应请求数据分组同时到达接收节点。通过获取并发窗口数从而计算出交换机缓存队列的长度或者瓶颈链路中剩余带宽的大小，作为调整数据源端发送速率大小的关键参数。然而在实际环境中，由于中间节点排队时延以及传输路径的差异，N 个数据源端发送的请求数据分组同时到达最后一跳节点的概率非常小，以此为计算基础所带来的误差不可避免。因此，本文采用的方法是基于排队论对数据流的到达行为进行建模，从而设计更为合理的发送窗口调节机制。

3.2.1　数学模型

目前，数据流到达行为的研究主要针对 TCP 流，所采用的研究方法主要分为 2 种：一种是基于长时间粒度统计发现 TCP 流具有自相似[18]或者长相关的特性[19]，对此特性进行具体研究；另一种是根据排队论的相关理论进行研究，例如文献[20]通过实际测量和仿真分析指出在链路带宽足够的情况下，数据流的到达行为服从泊松分布。由于数据中心链路具有高带宽、低时延特性，因此更适合采用排队论对数据流的到达行为进行数学建模并分析[21]。

文献[1]指出，当前数据中心普遍使用浅缓存的商用以太网交换机，此类交换机的特点是采用共享缓存交换结构。共享缓存结构是指交换机所有的输出和输入端口都共享一个缓存池，并且所有经过交换机转发的数据分组都需要在缓存中存储转发，那么一台交换机就可以抽象成一个服务窗口，此外，可以认为交换机对数据流的转发即对数据流的服务时间服从指数分布。由于算法被触发时，控制器需要立即响应，因此需要对源端发送速率进行调整，假设 t 时刻算法执行时需要数据源端减小数据分组进入系统的概率，那么数据流的到达率就不再是稳定值，而是依赖 t 时刻缓存队列长度 k 的函数，因此可以采用基于可变到达率的 G/M/1/∞排队模型进行建模分析[22]，图 1 描述了具有可变到达率的数据流生灭过程。

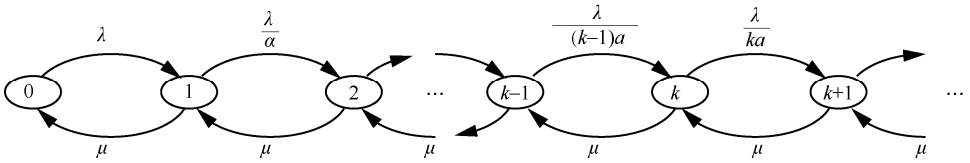


图 1 可变到达率的数据流生灭过程

以图 1 为例，假设缓存队列长度为 k（$k \geqslant 1$），数据流以 $a_k = \frac{1}{ak}(a \geqslant 1)$ 的概率进入排队系统，λ 为到达率，μ 为交换机服务速率。可以得到该生灭过程稳态下的数据流到达概率分布函数，如式(2)所示。

$$P_k = \frac{\rho^k}{(k+1)!(1+\rho e^{\frac{\rho}{a}})} \tag{2}$$

其中，P_k 为 t 时刻队列长度处于 k 状态的概率分布，ρ 为数据流的排队强度。

3.2.2 优化问题

当 t 时刻算法触发时，交换机缓存中队列长度超过设定阈值，控制器需要计算出数据源端合适的发送窗口大小以防止缓存溢出。此时，t 时刻超出阈值部分的队列长度为

$$G(t) = [Q(t) - K_t]^+ \tag{3}$$

其中，$Q(t)$为 t 时刻交换机缓存队列长度；K_t 为所设队列长度阈值；$[\bullet]^+$表示正值，保证优化问题有意义。那么优化问题就是使式(3)的队列长度差值 $G(t)$ 最小，优化问题的目标函数为

$$\min G(t) \tag{4}$$

排队系统进入稳定的工作状态时与时刻 t 无关，因此式(4)优化的目标函数 $G(t)$可以变形为

$$G(K) = \sum_{K=K_t}^{\infty} (K - K_t) P_k \tag{5}$$

其中，K 为当前队列长度。将式(2)代入式(5)并对其求和，整理后得到式(6)。

$$G(K) = \sum_{K=K_t}^{\infty} (K - K_t) P_k = \sum_{K=K_t}^{\infty} \frac{(K - K_t)\rho^k}{(k-1)! a^{k-1} \left(1 + \rho e^{\frac{\rho}{a}}\right)} = \frac{\rho e^{\frac{\rho}{a}}}{(1+\rho e^{\frac{\rho}{a}})} \left(\frac{\rho}{a} + 1 - K_t\right) \tag{6}$$

其中，$e^{\frac{\rho}{a}}$ 的取值范围为[0, 2.7]，并用常数符号 C_1 代替；$\rho = \frac{\lambda}{\mu}$；到达率 λ 可以用一个 $\overline{\mathrm{RTT}}$ 内发送窗口值表示，即当发送端以速率 r_i 向链路注入数据流时，$\lambda = r_i \overline{\mathrm{RTT}}$。假设交换机转发分组能力即交换机服务速率 μ 值固定，可以设 $C_2 = \frac{\overline{\mathrm{RTT}}}{\mu}$。对式(6)进一步变换后，得到的目标函数如式(7)所示。

$$G(r_i) = \frac{C_1 C_2^2 r_i^2 + \alpha C_1 C_2 (1 - K_t) r_i}{\alpha C_1 C_2 r_i + \alpha} \tag{7}$$

优化问题总是伴随着约束条件。首先，链路实际负载不能超过链路自身承载能力，链路负载能力用 C_l 表示，即 $r_i^l \leqslant C_l$；其次，优化问题变量的非负取值约束，即 $r_i > 0$。最终的优化问题为

$$\min \sum_i \frac{C_1 C_2^2 r_i^2 + \alpha C_1 C_2 (1 - K_t) r_i}{\alpha C_1 C_2 r_i + \alpha}$$
$$\text{s.t } r_i^l \leqslant C_l,\ r_i > 0 \tag{8}$$

式(6)是目标函数的代数变化形式。设 $\delta = \frac{\rho}{\alpha} + 1 - K_t$，如果 $\delta < 0$，说明不需要对优化目标函数进行求解，即不需要对源端发送速率进行限速；否则，求解式(8)的优化问题得到数据源端发送速率的目标解。由于式(8)的优化问题属于非线性比式和的分式规划，是一类全局优化问题，求解该类优化问题已被证明属于 NP-hard 问题[23]，可采用分支定界法对该类问题进行求解。最终，将优化问题转换为控制器端可执行的算法，通过求解获取数据源端的目标速率进一步确认数据源端的目标发送窗口值，即发送窗口调节算法（SWAA，sending window adjusting algorithm），如算法 1 所示。

算法 1 SWAA

1) #function definition

2) def modCwnd (r_i, $\overline{\mathrm{RTT}}$) /*发送窗口修改函数*/

3) #begin

4) new_cwnd　/*定义修改后的发送端窗口变量*/

5) T　/*设定控制器轮询周期值*/

6) B_T　/*轮询周期内数据流传输字节值*/

7) $K_t > \dfrac{C \times \overline{\mathrm{RTT}}}{7}$　/*设定队列阈值*/

8) $\overline{\mathrm{RTT}}$ = sampled_rtt / sampled_num /*计算链路平均时延*/

9) $r_i^0 = \dfrac{B_T}{T}$　/*计算数据流 i 初始速率*/

10) $\delta = \dfrac{\rho}{a+1} - K_t$

11) if $\delta < 0$

12)　return null

13) else

14)　C_1= random（0, 2.7）/*取值范围为[0, 2.7]*/

15)　$C_2 = \dfrac{RTT}{\mu}$

16)　求解优化问题式(8)获取源端目标速率 r_i

17)　new_cwnd = modCwnd(r_i, $\overline{\mathrm{RTT}}$)

18)　return

19) end if

20) #end

步骤 2)定义了发送窗口修改函数，参数是优化后的数据源端目标速率值 r_i 以及链路的平均时延 $\overline{\mathrm{RTT}}$。步骤 3)~步骤 20)具体实现了数据源端目标速率调节机制，其中，步骤 8)链路的平均时延由时延抽样值 sampled_rtt 和抽样次数 sampled_num 确定；步骤 9)通过控制器轮询周期及轮询周期内统计到的数据流计数器中的传输字节值计算出数据流 i 初始速率，从而获得该流的到达率；步骤 17)通过调用发送窗口修改函数，得到数据源端的发送窗口目标值并返回存储，控制器通过将 new_cwnd 重新写入从反馈回路获取到的 TCP_ACK 分组，最终达到调节数据源端速率的目的。

4　算法实现

当前的研究工作中，通常依靠 TCP 连接建立的反馈回路传递拥塞链路信息或者拥塞窗口大小调节信息，以达到调节发送速率的目的，避免链路拥塞的发生。但是，如果反馈回路拥塞或者发生故障则严重影响算法的时效性。本文 SSRC 算法能够利用 SDN/OpenFlow 架构的优势，很好地解决当前研究算法时效性不高的问题。首先，通过对链路中出现的长流以及交换机队列长度的监测，快速定位拥塞可能出现的节点位置并触发算法；控制器利用 TCP 会话建立的反馈回路修改接入层交换机的反向流表匹配规则，极大地提高数据源端对拥塞节点的响应时间，提高算法的时效性，在保证短流的传输截止时间的同时，防止出现 Incast 问题。算法流程如图 2 所示。具体步骤如下。

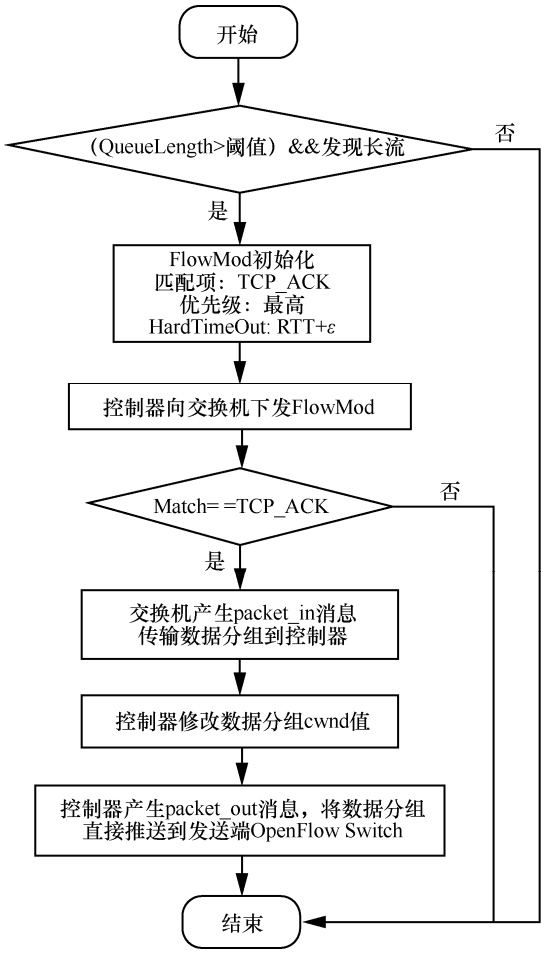


图 2　算法设计流程

1) 控制器定时向交换机查询流表计数器值，同时监控交换机缓存队列长度。当计数器值大于长流阈值（判断为长流），且队列长度大于设定阈值时，触发算法。

2) 控制器向接收端接入交换机下发 FlowMod 命令。该 FlowMod 关键参数如下：匹配项为 TCP_ACK，优先级为最高，HardTimeOut 为 RTT+ε（ε<RTT）。

3) 在接入层交换机上，如果数据流与更新后的

流表匹配成功，则将数据分组发送到控制器。

4) 控制器收到 packet_in 消息，判断其 reason == OFPR_ACTION 后，修改数据分组 cwnd 值。

5) 将修改后的 ACK 分组，通过 packet_out 消息，直接推送到数据流源端的接入层交换机。

步骤 2)的操作是由于数据源端速率更新后，至少在一个 RTT 时间后接收端才能收到更新的数据分组，因此，在 RTT+ε 内都应保持 SSRC 更新后的发送窗口值。

将上述步骤转换成控制器端可执行的算法程序，并提出数据源端控制算法 SSRC（SDN-based source rate control），如算法 2 所示。

算法 2 SSRC

```
1) #initialization
2)  QueueThresh = 10; LongFlowThresh = 100;
3)  LongFlow = false; QueueLengthExceed = false;
4) #begin
5) function IfLongFlow()
6)  Packet_match_counter = Controller.GetPerFlowStatistic();
7)  if Packet_match_counter > LongFlow-Thresh then
8)  LongFlow = true;
9)  end if
10)  return LongFlow
11)end function
12)function IfQueueLengthExceed()
13)  QueueLength = Switch.GetPortStatistic();
14)  if QueueLength > QueueThresh then
15)  QueueLength = true;
16)  end if
17) end function
18) if LongFlow && QueueLengthExceed then
19)  flow_modification = true;
20)  FlowMod.setMatch(TCP_ack);
21)  FlowMod.setPort(OFPort.controller);
22)  FlowMod.setPriority(65536);
23)  FlowMod.setHardTimeOut(RTT + ε)
24) end if
25) if packet_in  then
26)  if packet_in_reason == OFPR_ACTION then
27)  IPv4 Ipv4Header = (IPv4) eth.getPayLoad();
28)  TCP tcp = (TCP) Ipv4Header. GetPayload();
29)  short cwnd = tcp.get WindowSize();
30)  SWAA(cwnd);
31)  OFPacketOut.Builder OutputChangedPacket;
32)  end if
33) end if
34) #end
```

步骤 1)~步骤 4)设置了 SSRC 算法的初始值。步骤 5)~步骤 11)定义了长流判断函数。步骤 12)~步骤 16)定义了队列长度是否超过设置阈值的判断函数。步骤 18)~步骤 33)是 SSRC 的核心代码，其中，步骤 18)~步骤 24)判断当目标长流出现并且队列长度超出阈值，也就是步骤 18)的判断为真时，控制器会向交换机下发 FlowMod 的命令，即步骤 19)~步骤 23)代码；步骤 25)~步骤 33)执行了当控制器接收到 packet_in 消息时，判断出产生 packet_in 的原因是 FlowMod 匹配项得以匹配，此时控制器会对数据分组进行解析，获得数据分组头中的拥塞窗口值，调用 SWAA 算法进行修改（步骤 30)），并将修改后的数据分组直接推送到源端的边缘层交换机。至此，SSRC 执行完毕。由于算法 2 是在一段时间内遍历 n 条数据流并进行长流的判定，因此算法 2 时间复杂度为 $O(n)$。

5 仿真与评估

5.1 仿真平台构建

集中架构的仿真平台采用 NS3+Floodlight 进行搭建。平台运行的宿主机是戴尔 OptiPlex9020 服务器，设备硬件的性能参数为：8 核/3.4 GHz 主频的 64 位处理器，10 GB 内存，操作系统采用 Ubuntu16.04 版本。同时，宿主机部署了支持对 OpenFlow 协议分析的 wireshark 软件。

NS3 仿真器采用 v3.6 版本，使用 Floodlight 控制器作为外部控制器，并通过 Tapbridge 与 NS3 相连。由于 NS3 具有离散时间仿真的特点，即一旦仿真开始，就不能中途修改参数。为了实现 SSRC 的控制功能，本文编写 2 个功能模块预置在 Floodlight

的应用程序中：一个是 AddDSCPFlowMod，实现下发流表和添加 meter entry；另一个是 DSCPController，实现解析数据分组和修改发送窗口值后，将携带发送窗口目标值的数据分组直接推送到连接发送端的接入层交换机。

仿真拓扑选择当前数据中心普遍采用的以交换机为核心的多层拓扑结构，实验将构建 K（交换机接入端口数）值可变的胖树（fat-tree）拓扑，并以 K=8 即具有 8 个 POD（performance optimization datacenter）的拓扑规模进行仿真实验，如图 3 所示。其中，核心层的交换机编号为 S101~116，汇聚层的交换机编号为 S201~S232，边缘层的交换机编号为 S301~S332，终端主机的编号为 H001~H128。

5.2　实验设计

5.2.1　实验对象选择

SSRC 性能实验的对比算法选择 TCP_NewReno 以及先前研究工作中具有代表性的 2 个解决算法。其中，代表终端解决算法采用文献[3]的方法，即减小最小超时重传时间 RTO_{min}，该算法依然基于 NewReno 算法，只是减小了 RTO_{min} 值，实现方法简单，易于部署；DCTCP 则代表交换设备解决算法，基于 ECN 的标记功能，但是不同于 ECN 对交换机平均队列长度阈值做出响应，DCTCP 是对交换机瞬时队列超过阈值的数据分组进行标记，其次，ECN 的发送端在收到接收端标记的响应数据分组后，发送窗口减半，而 DCTCP 发送端通过感知网络中间节点的拥塞程度来动态调节发送窗口大小，具体做法是被标记的分组数量越多、衰减因子越大，发送窗口就越小。相比 ECN，DCTCP 对网络拥塞的响应更加及时并且能保证网络吞吐量的需求，是针对 Incast 问题比较有效的解决算法。

5.2.2　关键变量设置

考虑到 NS3 仿真平台与实际部署的差距，设计实验时首先需要对仿真过程中的关键变量（即会对实验结果产生重要影响但不是实验研究对象的变量）进行设定，关键变量设置的合理性将直接关系到实验结果的准确性。本实验需要设置的关键变量是仿真系统默认的 RTO_{min} 值以及背景流个数（长流）。

1) RTO_{min} 值

NS3-3.26 版本内核的默认 RTO_{min}=1 s，但是在实际实验过程中发现了虚假重传的现象。证明实验如下。20 个数据源端在没有背景流的条件下同时发送请求短流，并保证足够的链路带宽以及数据接收端的缓存容量完全可以容纳所有的数据分组。通过对实验数据统计发现并没有出现分组丢失现

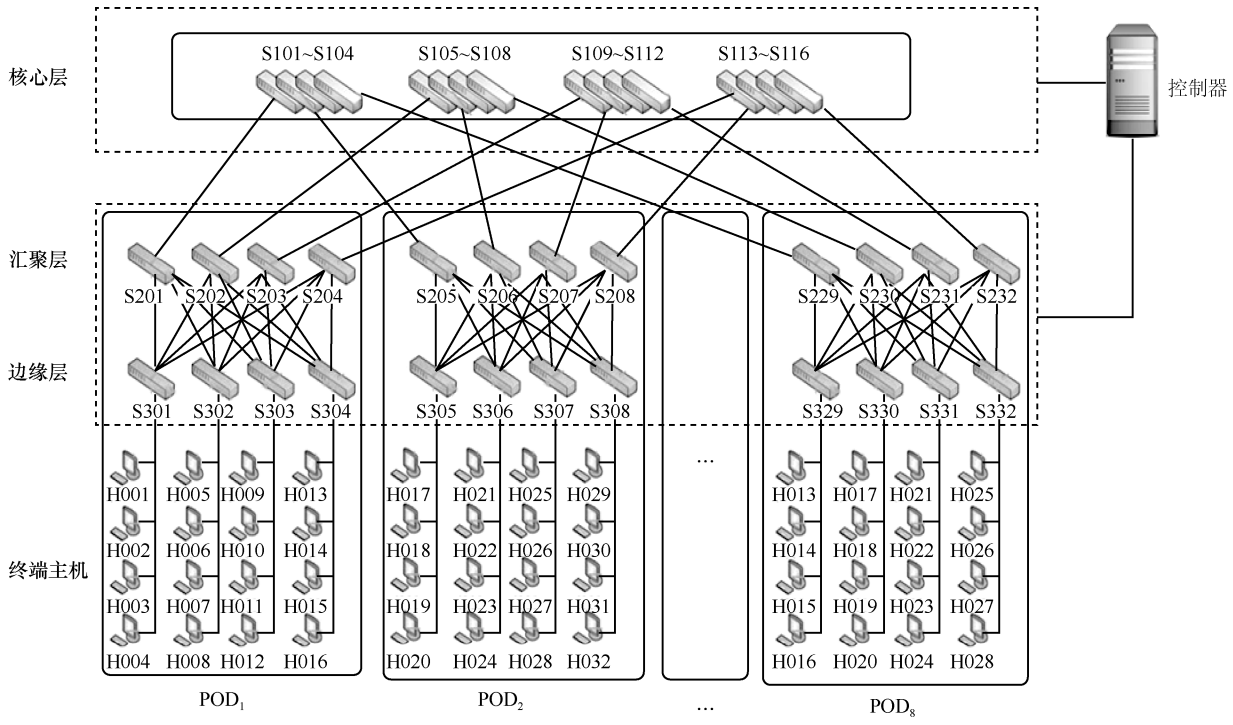


图 3　实验拓扑

象，然而经过 wireshark 抓取分组分析却发现了虚假重传现象，如图 4 所示。当 RTO_{min}=1 s 时，所有发送短流都存在虚假重传。因此，为了准确还原对比算法的实验效果，仿真中针对 RTO_{min} 参数值的设置需要考虑 2 个关键要素：首先，需要模拟出 DCTCP 运行的默认 RTO_{min} 值，既能保证不发生虚假重传，又能保证不会因为 RTO_{min} 值过大而增大时延；其次，需要模拟出 Linux 内核中默认的 RTO_{min}=200 ms 的 TCP 连接传输效果。为了满足以上目标，实验重新设计如下：20 个主机同时发送请求短流，在没有背景流的情况下，以 0.1 s 为步长改变 RTO_{min}，统计这 20 条数据流在不同 RTO_{min} 值下发生虚假重传的比例。实验结果如图 4 所示，可以观察到当 RTO_{min}<3.1 s 时，所有的数据流都会发生至少一次的虚假重传，随着 RTO_{min} 值增加，发生虚假重传的百分比减少，当 RTO_{min}>5 s 时，基本达到稳定值，即所有数据流都不会受到虚假重传的影响。

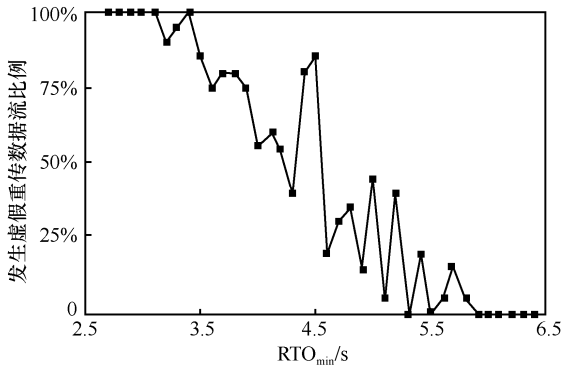


图 4 最小超时重传时间设置分析

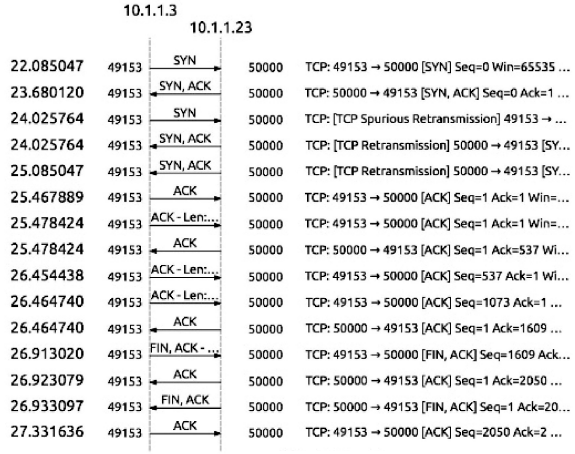
为了保证实验结果的准确性以及还原比较对象原本的实验效果，需要进一步确认 RTO_{min} 值。从图 4 的分析中可以观察到，RTO_{min}≥5.8 s 时性能比较稳定，通过多次测试，最终确认 RTO_{min}=10 s。为了验证合理性，以默认值 10 s 为基准，比较了当 RTO_{min} 设置过小的情况下虚假重传的表现。图 5 显示了主机 10.1.1.3 在 RTO_{min}= 1 s 和 RTO_{min} = 10 s 下的实验结果对比。

从图 5 可以观察到，当 RTO_{min}=1 s 时，在 TCP 通信连接建立的过程中，所有数据分组几乎同时在 22 s 左右发送，导致数据流时延增大，而此时的 RTO_{min} 值又太小，因此导致在 24 s 源端又发送了一次连接请求。当 RTO_{min}=10 s 时，可以明显看到源端 3 次握手后顺利地建立了连接，缩短了流传输的时间。

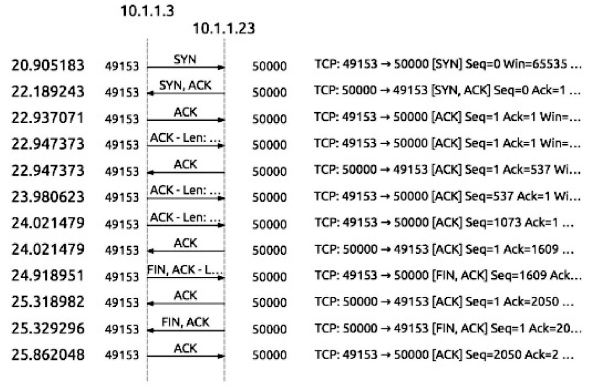
此外，实验也模拟了 RTO_{min} 设置过大的情况，并以保证 Linux 内核中默认 RTO_{min}=200 ms 时的 TCP 并发连接时的网络性能表现（吞吐量下降 2 个数量级），测试后发现 RTO_{min}=100 s 时可以满足对比要求。最终，对比实验结果发现，当 RTO_{min}=10 s 时，流完成时间为 6.3 s，当 RTO_{min}=100 s 时，流完成时间变成了 103.4 s，因此本实验设置 RTO_{min}=10 s 能够模拟出真实的网络情况。

2) 背景流个数

在以往研究工作的同类实验中，一般选择背景流为 5 条。但是实际应根据实验规模来确定，因为背景流数目太大，占总数据流的比重过大，一方面会加剧请求短流的分组丢失；另一方面



(a) RTO_{min}=1 s



(b) RTO_{min}=10 s

图 5 RTO_{min} 分别为 1 s 和 10 s 设置分析

在模拟 Incast 环境时，即使请求短流并发数目很小也会有严重的分组丢失现象。背景流数目太少又无法提供客观的比较值，在计算吞吐量时会有较大误差。因此应选择与实际环境中请求短流和背景流比例相匹配的数据流个数为宜，具体的比例参见 DCTCP[1]。图 6 是在并发服务器个数分别为 20 和 30 下，改变背景流个数，对 20 条请求短流完成时间的统计结果，目标为尽量减小背景流数目变化对不同并发数下流完成时间的影响，最终选定本次实验的背景流数为 6 条。

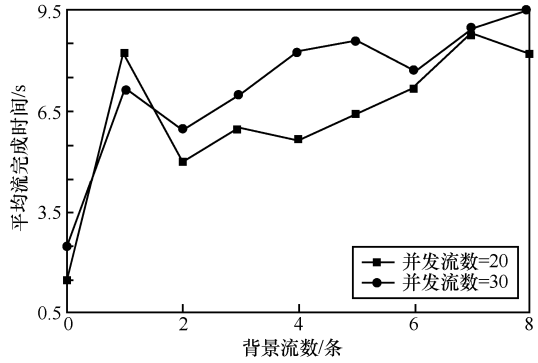


图 6　背景流数量设置分析

5.2.3　实验部署

通过对实验关键变量的分析与设置，本文实验部署如下：实验瓶颈链路带宽设置为 100 Mbit/s，网络节点交换机的缓存容量为 64 KB。实验拓扑如图 3 所示，每个 POD 连接 16 台服务器主机，共 8 个 POD，编号为 1~8。数据发送端和接收端属于不同 POD，故设定 POD_8 中一台主机为接收端，同时为了避免服务器接入链路成为瓶颈链路，发送端主机由其余 POD 平均分配，其中，POD_1~POD_6 分别随机选择一台主机，以随机时间依次开始向接收端发送的长流作为背景流量；每个 POD 另外选择 1~12 台主机并发产生 SRU，每个 SRU 大小设为 256 KB，因此接收端共计请求数据源端发送流量大小为 N×SRU，N 为并发数，$1 \leqslant N \leqslant 80$。SSRC 的性能表现将基于数据流完成时间、网络平均吞吐量以及分组丢失率这 3 个指标进行评估。

5.3　性能评估

5.3.1　数据流完成时间

由于 NS3 仿真平台的特殊性，在实际环境中，平均流完成时间正比于并发服务器个数，系数为 $\frac{\text{FlowSize}}{\text{链路带宽}}$。同时，考虑到 NS3 平台中使用了 CSMA 信道，以及其他可能的干扰因素，为了保证实验的严谨性，进行图 7 (a)所示的实验，找出在没有背景流的情况下，增加并发服务器个数与数据平均流完成时间拟合的二次函数。图 7(b)为 4 种算法在增大并发服务器个数下的表现。首先，由于背景流的存在，所有算法下短流的传输时间都受到了很大的影响。例如比较并发数为 45 条，增加 6 条背景流，共 51 条数据流同时竞争信道时，短流的平均流完成时间最优可达到 10.50 s（SSRC、NewReno 下为 30.12 s），而没有背景流存在时，45 条数据流传输的平均流完成时间为 5.33 s，也就是说出现了成倍的增长。此外，对比 4 种算法的表现可以得出，并发数较少时，不同算法表现没有太大差异，流完成时间基本维持在 7 s 左右；随着数据流并发数的增多，流完成时间开始增加。相比于其他 3 种算法，SSRC 增加幅度为 62.3%，NewReno、RTO_{min} 和 DCTCP 分别为 411.4%、273.2%和 168.1%，SSRC 的优化效果十分明显。

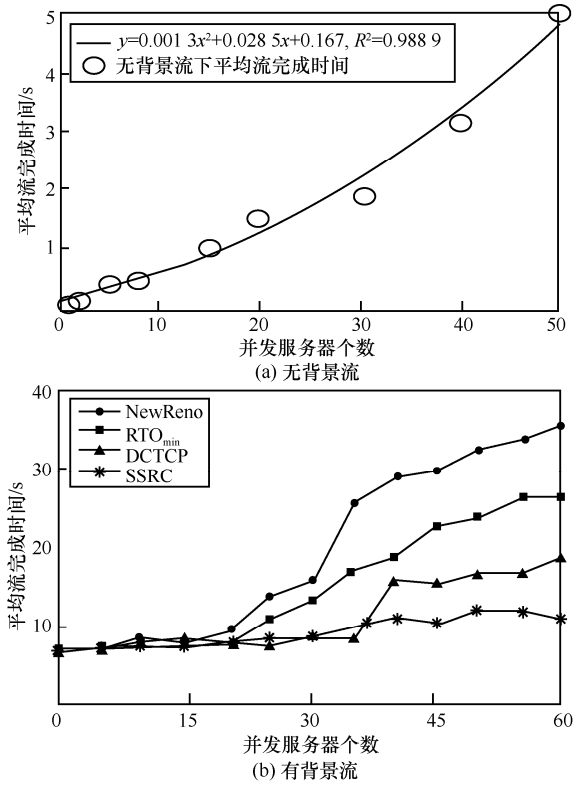
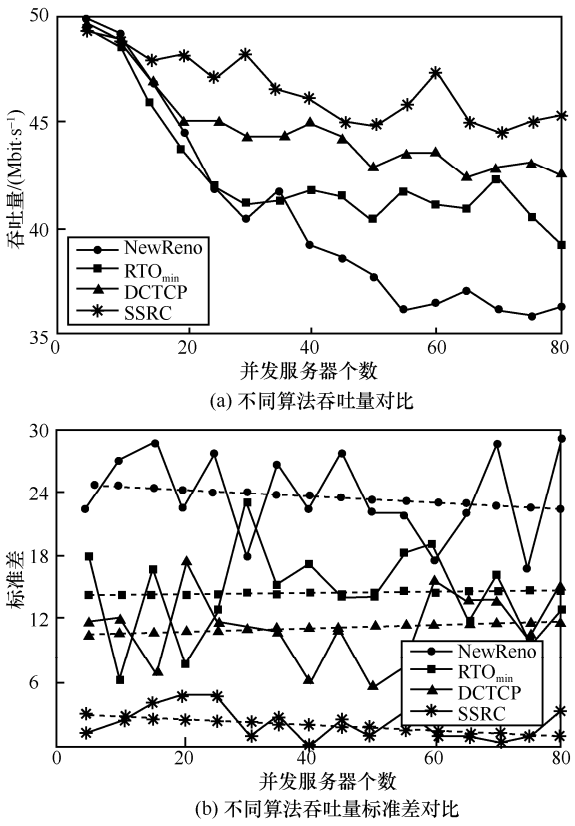


图 7　流完成时间比较

5.3.2　网络平均吞吐量

吞吐量主要针对长流来分析。长流对于时

延不是十分敏感，但是对于吞吐量的要求很高，当链路发生拥塞时，长流分组丢失会造成吞吐量的断崖式下降。另一个特征是 TCP NewReno 下不同背景流的吞吐量方差很大，反映出部分长流在传输过程中的分组丢失可能更多，而一旦发生分组丢失，很难再和其他正常传输的数据流竞争信道，最终表现为链路资源分配的不均匀。因此实验就上述 2 个方面进行比较。由图 8(a)可以看出，相比于其他算法，随着并发数的增多，SSRC 依然能保持较高的吞吐量；图 8(b)进一步从吞吐量的标准差对比来验证 SSRC 性能。图 8(b)中不同的虚线是对不同并发数下吞吐量标准差的一次线性拟合，可以清晰地看出，通过 SSRC 对源端速率的精确控制，能够保证链路充分利用，并且较为平均地将资源分配给每一条数据流。



(a) 不同算法吞吐量对比

(b) 不同算法吞吐量标准差对比

图 8　吞吐量性能比较

5.3.3　分组丢失率

分组丢失对于长流会造成吞吐量的明显下降，对于时延敏感的短流会造成超时重传。为了明确 NS3 中长流和短流在并发时各自的特性，设计了以下实验：数据中心拓扑中维持 6 条背景流，在仿真开始后的 23 s 左右（此时背景流传输稳定），30 个服务器并发发送大小为 2 KB 的 SRU，统计在这 30 个服务器第一个分组发送后，每秒内长流和短流各种传输的数据分组数目，以及分组丢失的数目（在 RTO_{min} 算法下）。仿真实验中，第一个短流的数据分组在 23.484 645 s 发出，最后一个数据分组在 35.837 998 s 收到，统计后得到的结果如图 9 所示。

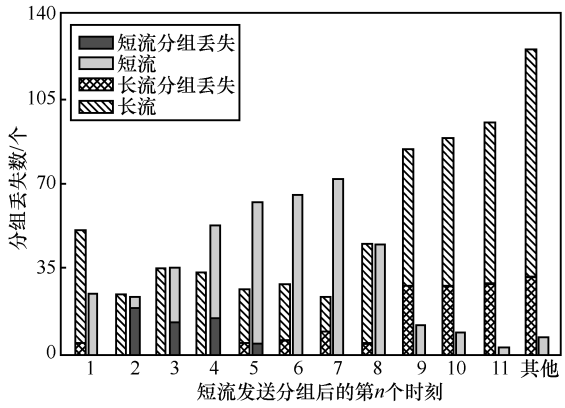


图 9　长短流分组丢失比较

短流在开始的 4 s 内，发生了严重的分组丢失现象，而长流的分组丢失不严重，说明在开始竞争信道时，交换机缓存中基本上都是长流的数据分组，后来到达的短流很容易超过阈值而被丢弃，也就是说短流在与长流的竞争中处于劣势。5~8 s，链路依然繁忙，此时已经有部分的短流占据了缓存，导致之后的长流分组丢失率上升，而且短流本身只有 2 KB，很容易塞满交换机而不被丢弃。9 s 后短流基本已经传输完成，通过交换机的数据分组占比快速下降，即使缓存溢出，被丢弃的概率也很小。此外，整个实验结果都反映了长流的“贪婪性”，因为如果链路资源均匀分配，每个时间段内发送的数据分组平均应该有 16.67%来自长流，而实际中长流占比最少时（第 7 s）也达到了总传输量的 33.33%，而且长流占比的下降很可能是之前连续的分组丢失使源端退避而暂时降低了链路资源的占用。通过上面的实验可以看出，NS3 环境下长短流基本的特性和实际网络环境中一致。在长短流特性已知的前提下保证背景流数目为 6 且不变，增加并发服务器个数，对短流的分组丢失率进行统计，如图 10 所示。对比发现，SSRC 的控制算法降低分组丢失率的表现优越，即使在链路状态十分恶劣的情况下，也基本能够保证分组丢

失总数在 10 个以内。

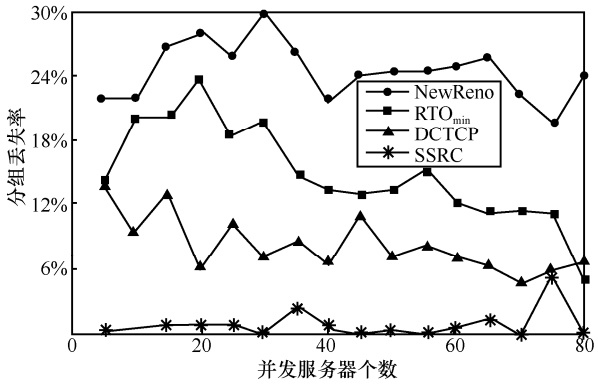


图 10　分组丢失率比较

6　结束语

本文针对数据中心网络如何保证时延敏感流的传输时间问题进行了相关研究，在 SDN/OpenFlow 的架构下，提出了一种基于数据源端速率控制的算法 SSRC。该算法能够准确定位可能发生拥塞的节点设备，通过控制器快速进行应对策略的响应，相较于传统网络中接收端通过反馈回路进行数据源端速率调节的算法，能够极大地缩短响应时间，解决现有算法时效性不高的问题。最后，通过将 NS3 仿真工具与 Floodlight 外部控制器相连，实现 SDN/OpenFlow 架构下的数据中心网络环境中进行，同时与 DCTCP 等 3 种解决算法进行比较，仿真实验结果证明 SSRC 能够保证时延敏感流的传输时间，同时能够很好地解决 Incast 问题。

参考文献：

[1] ALIZADEH M, GREENBERG A, MALTZ D A, et al. Data center tcp (dctcp)[C]//The ACM Conference on SIGCOMM. ACM, 2010: 63-74.

[2] DEAN J, GHEMAWAT S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[3] XU L, XU K, JIANG Y, et al. Enhancing TCP incast congestion control over large-scale datacenter networks[C]//The IEEE International Symposium on Quality of Service. IEEE, 2016: 225-230.

[4] SHAH Z. Mitigating TCP incast issue in cloud data centres using software-defined networking (SDN): a survey[J]. KSII Transactions on Internet and Information Systems, 2018, 12(11): 5179-5202.

[5] ABDELMONIEM A M, BRAHIM B, ABU A J. Mitigating incast-TCP congestion in data centers with SDN[J]. Annales des Telecommunications/Annals of Telecommunications. 2018, 73(3):263-277.

[6] 杨洋, 杨家海,温皓森. 基于时隙传输的数据中心路由算法设计[J]. 软件学报. 2018. 29(8): 2485-2501.
YANG Y, YANG J H, WEN H S. Another routing algorithm design based on timeslot of transmission in data center networks[J]. Journal of Software, 2018, 29(8):2485-2501.

[7] ZHANG J, REN F Y, LIN C, et al. Modeling and solving TCP incast problem in data center networks[J]. IEEE Transactions on Parallel and Distributed Systems, 2015, 26(2):478-491.

[8] VAMANAN B, HASAN J, VIJAYKUMAR T. Deadline-aware datacenter TCP (D2TCP)[C]//The ACM Conference on SIGCOMM. ACM, 2012: 115-126.

[9] WU H, FENG Z, GUO C, et al. ICTCP: incast congestion control for TCP in data center networks[J]. IEEE/ACM Transactions Networking, 2013, 21(2): 345-358.

[10] MORTEZA K, WAKEMAN I, GEORGE P. MMPTCP: a multipath transport protocol for data centers[C]//The IEEE International Conference on Computer Communications. IEEE, 2016.

[11] FORD C, RAICIUM, HANDLEY S, et al. RFC6824: TCP extension for multipath operation with multiple addresses[R]. IETF, (2013-01-01) [2018-12-20].

[12] RAMAKRISHNAN K, FLOYD S, BLACK D. RFC 3168: The addition of explicit congestion notification (ECN) to IP[R]. IETF, (2001-09-01) [2013-03-02].

[13] WILSON C, BALLANI H, KARAGIANNIS T, et al. Better never than late: meeting deadlines in datacenter networks[C]//The ACM Conference on SIGCOMM. ACM, 2011: 50-61.

[14] AMAR P, ELIE K, VIJAY V, et al. Measurement and analysis of TCP throughput collapse in cluster-based storage systems[C]// The USENIX Conference on File and Storage Technologies. USENIX Association, 2008.

[15] PAN R, PRABHAKAR B, LAXMIKANTHA A. QCN: quantized congestion notification an overview[R]. IEEE, (2009-05-29) [2019-01-28].

[16] ZHANG J, REN F Y, TANG L, et al. Taming TCP incast throughput collapse in data center networks[C]//The IEEE International Conference on Network Protocols. IEEE, 2013.

[17] WASSERMA M, HARTMAN S. OpenFlow switch specification[R]. Huawei, (2012-08-01) [2013-04-02].

[18] UHLIG S. Non-stationarity and high-order scaling in TCP flow arrivals：a methodological analysis[J]. ACM SIGCOMM Computer Communications Review, 2004, 43(2): 9-24.

[19] ROUGHAN M, VEITCH D. Measuring long-range dependence under changing traffic conditions[C]//The Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE, 1999: 1513-1521.

[20] BONALD T, COMTE C. The multi-source model for dimensioning data networks[J]. Computer Networks, 2016,109 (2): 225-233.

[21] CAO J, XIA R, YANG P, et al. Per-packet load-balanced, low-latency routing for clos-based data center networks[C]//The Seventh Conference on Emerging Networking Experiments and Technologies. ACM, 2013:49-60.

[22] 陆传赉. 排队论[M]. 北京:北京邮电大学出版社, 2009.
LU C J. Queuing theory[M]. Beijing: Beijing University of Posts and Telecommunications Press, 2009.
[23] SIEGFRIED S, JIANMING S. Fractional programming: the sum-of-ratios case[J]. Optimization Methods and Software, 2003, 18(2): 219-229.

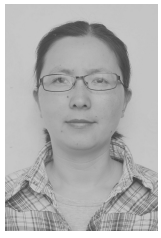
[作者简介]

杨洋（1980− ），男，江苏无锡人，博士，国防科技大学讲师，主要研究方向为计算机网络、路由协议、流量工程等。

曹敏（1985− ），女，陕西咸阳人，国防科技大学讲师，主要研究方向为通信与信息系统、信号处理等。

杨家海（1966− ），男，浙江云和人，清华大学教授、博士生导师，主要研究方向为计算机网络、网络管理与测量、网络安全、云计算与大数据等。

车崧（1979− ），女，甘肃兰州人，国防科技大学副教授，主要研究方向为信号与信息处理、图像通信技术等。



刘伟（1982− ），女，安徽安庆人，博士，国防科技大学副教授，主要研究方向为信号与信息处理、无线通信技术等。