基于平衡点的自适应 RED 算法*

李风华[1+], 卢向群[2], 吴建平[3]

[1](清华大学 计算机科学与技术系,北京 100084)

[2](北京邮电大学 计算机科学与技术学院,北京 100876)

[3](清华大学 计算机科学与技术系,北京 100084)

An Adaptive RED Algorithm Based on Equilibrium Point

LI Feng-Hua[1+], LU Xiang-Qun[2], WU Jian-Ping[3]

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

[2](School of Computer Science and Technology, Beijing University of Post and Telecommunications, Beijing 100876, China)

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

+ Corresponding author: Phn: +86-10-62772375, E-mail: lfh@star.bjnet.edu.cn

Received 2004-01-16; Accepted 2004-03-29

Li FH, Lu XQ, Wu JP. An adaptive RED algorithm based on equilibrium point. *Journal of Software*, 2004,15(Suppl.):36~44.

Abstract: Quality of services is a key to next generation Internet. More people focus on these problems, such as active queue management and random early detection. In this paper, we present an algorithm of adaptive RED, which can modify the max probability of packets dropping. We modify this parameter based on calculating the equilibrium point, so we can get the steady-state faster than some other methods based on self-adjustment.

Key words: active queue management; random early detection; probability of packets dropping; adaptive; equilibrium point

摘 要: 网络服务质量控制已经成为互联网技术的主要研究方向之一.主动式队列管理以及 RED(random early detection)算法的改进问题是近期网络服务质量研究的一个热点问题.提出的基于平衡点的自适应 RED 算法可以根据网络上的流量特点,动态地修改最大丢弃概率,能够对不同类型的数据流聚集进行调节,与目前大多数其他 RED 改进算法相比,该算法没有采用逐步逼进的途径,而是基于计算的平衡点,修正最大丢弃概率,可以更快的达到稳定状态,从而解决 RED 算法对于参数的依赖性和队列抖动问题.

关键词: 主动式队列管理;随机早期检测;丢弃概率;自适应;平衡点

AQM(active queue management)是近几年在网络服务质量控制研究方面的一个热点问题,RED(random

* Supported by the National High-Tech Research and Development Plan of China under Grant No.863-300-02-04-99 (国家高技术研究发展计划(863))

作者简介: 李风华(1973－),男,辽宁沈阳人,博士生,主要研究领域为计算机网络体系结构,服务质量控制,组播;卢向群(1973－),男,工程师,主要研究领域为计算机网络通信,软件工程;吴建平(1955－),男,教授,博士生导师,主要研究领域为计算机网络体系结构,协议形式化描述,协议测试.

early detection)[1,2]算法是主动式队列管理中一个比较有效的算法.它通过计算队列的平均队列长度和随机丢弃概率,对到达队列的分组进行早期随机丢弃,从而避免 FIFO(first in first out)队列存在的满队列(full queue)和队列闭锁(lock-out)问题.但是目前的 RED 算法还存在一定的问题,例如参数配置的合理化和队列长度的抖动问题.

本文通过对 RED 算法的仿真和分析,根据平衡点原理,提出了一种基于平衡点的自适应 RED 算法.该算法可以通过对队列的实时性监测,动态地调整队列最大丢弃概率,从而有效地克服了原算法对于参数配置的依赖性.通过基于 NS2 的仿真实验,结果证明了基于平衡点的自适应 RED 算法的有效性和可行性.目前国际上也有很多其他的关于 RED 算法的改进方法[3,4],但是大多数方法都是在原来最大丢弃概率的基础上进行修改,所以往往经过多个修正周期后才能达到稳定状态;而我们的算法是直接计算平衡点,根据平衡点的值来修改最大丢弃概率,因此可以比较快地达到稳定状态.

1 RED 算法及其存在的问题

FIFO 算法,即传统的先入先出队列,只要队列没满,那么到达队尾的分组就可以进入队列进行排队;如果队列已经达到满状态,那么到达队尾的分组就会被丢弃掉.一旦 FIFO 队列达到满状态,就可能出现两个问题.一是队列闭锁问题,它会导致带宽分配的不公平性,某些用户受到很好的服务,而某些用户却没有带宽可以使用.另外一个问题是满队列问题,一旦一个 FIFO 队列进入满状态,那么它至少要等待一个分组的服务时间之后才能开始接受新的分组,并且在这段时间里,所有到达队列的分组都会被丢弃掉.而多个数据流的连续多个分组被丢弃对于 TCP 这样具有流量控制功能的协议而言是致命的,这些数据流由于多个连续分组的丢失,而同时进入慢启动状态(slowstart),引起全局同步(global synchronization),导致协议的效率和链路利用率的严重下降.

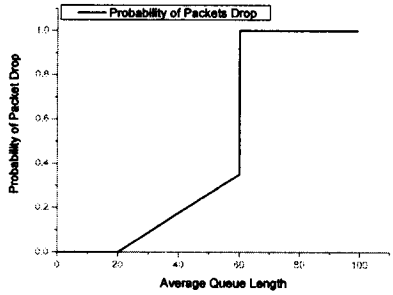

图 1 丢弃概率函数图像

由于上述 FIFO 队列的问题,所以在主动式队列管理中,避免队列进入满状态是一个非常重要的原则[5].RED 算法就是通过在队列到达满状态之前,通过随机早期丢弃分组来避免队列进入满状态.RED 算法随机早期丢弃的概率函数是平均队列长度的线性函数,丢弃概率函数参见公式(1)(函数图像见图 1).

$$p = H(\bar{q}) = \begin{cases} 0, & 0 \le \bar{q} \le \min_{th} \\ \dfrac{\bar{q} - q_{\min}}{q_{\max} - q_{\min}} P_{\max}, & \min_{th} < \bar{q} < \max_{th} \\ 1, & \max_{th} \le \bar{q} \le q_{\lim} \end{cases} \tag{1}$$

$$\bar{q} = (1-w)\bar{q} + wq \tag{2}$$

其中,p 为丢弃概率,$\bar{q}$ 为平均队列长度,q 为即时队列长度,w 为加权平均参数,$\max_{th}$ 和 $\min_{th}$ 分别为发生随机早期丢弃时队列平均长度的上下阈值(以下简称为丢弃上限和丢弃下限),$q_{\lim}$ 为队列的最大容量,$p_{\max}$ 为最大丢弃概率.该函数表明,平均队列长度是即时队列长度加权平均,当平均队列长度小于丢弃下限 $\min_{th}$ 时,说明该队列的拥塞程度较轻,所有到达队列的分组都可以进入队列,即丢弃概率为 0;当平均队列长度介于丢弃下限 $\min_{th}$ 和丢弃上限 $\max_{th}$ 之间时,说明该队列比较拥塞,队列控制开始计算丢弃概率,并依据丢弃概率随机丢弃到达队列的分组,避免拥塞情况加剧;当平均队列长度大于丢弃上限 $\max_{th}$ 时,说明队列已经非常拥塞,那么所有到达队列的分组都会被丢弃,避免队列进入满状态.

由公式(1)可知,如果 $P_{\max}$ 的值设置的比较大,则每次计算的丢弃概率偏大,那么就会导致队列在轻负载环境下的过分丢包,使得链路利用率降低;如果 $P_{\max}$ 的值设置的比较小,则每次计算的丢弃概率偏小,那么在网络负载较重的时候,丢弃概率不足以控制队列长度的增加,会导致队列进入满状态,从而引发分组的连续丢弃和 TCP 的全同步问题.由此可知,RED 算法参数的选择对 RED 的性能有比较大的影响,严重的时候会导致队列长度剧烈抖动和链路带宽的大量浪费,所以 RED 算法对于这些需要配置的参数具有较强的依赖性.

而我们提出的自适应RED算法就是要根据网络的负载情况,动态地修改P_{max},使得RED算法可以适应网络的负载.

2 基于平衡点的自适应RED算法

2.1 平衡点

TCP协议是目前互联网使用最为广泛的一种协议,单个TCP流的控制过程包括慢启动过程和拥塞避免过程,4.3BSD以后的reno TCP协议都支持快速重传和快速恢复[6,7].通过对单个TCP流在拥塞避免阶段的稳态分析,我们可以得到单个TCP流的分组丢失率p和发送速率R(参见公式(3)):

$$p=\frac{8}{3w^2},\ R=\frac{\frac{3}{8}w^2S}{\frac{1}{2}wT}=\frac{3wS}{4T}=\frac{3S\sqrt{\frac{8}{3p}}}{4T}=\frac{\sqrt{\frac{3}{2}}S}{T\sqrt{p}} \tag{3}$$

其中p为分组丢失率,w为稳态的发送窗口大小,T为TCP流的RTT(round trip time)时间.我们假设在一个TCP流聚集里面所有流的RTT时间相同,流的数目为N,并且这些流平均分配路由器端口的带宽B.即每个流的发送速率R为:

$$R=\frac{B}{N} \tag{4}$$

而对于任意一个TCP流,它的RTT时间T包括两部分:在某一节点内部队列的排队时间T_{queue}和节点外的传输延迟T_0(参见公式(5)).

$$T=T_{queue}+T_0=\frac{q}{B}+T_0 \tag{5}$$

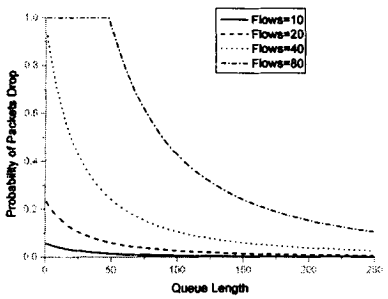

图2 不同流数量的G函数图像

因此,当队列以端口速率输出时(即输出端口带宽利用率为100%),我们可以得到即时队列长度关于分组丢失率的函数,(参见公式(6)和图2),u为端口利用率:

$$q=G(p)=\begin{cases}0 & 0\le u<1\\ \frac{\sqrt{\frac{3}{2}}SN}{\sqrt{p}}-BT_0 & u=1\end{cases} \tag{6}$$

那么当该TCP流聚集通过RED队列达到稳定状态时,我们可以通过方程组(参见方程组(7)):

$$\begin{cases}p=H(q)\\ q=G(p)\end{cases} \tag{7}$$

求解得到RED队列在稳态时的平均队列长度和随机丢弃概率,我们把该点定义为RED算法的平衡点[8].通过图2和图3可以得知,具有不同流量参数的TCP流聚集曲线和丢弃概率曲线的交点(即平衡点)是不同的:当交点对应的队列长度处于丢弃上限和下限之间时,TCP的流量控制功能可以使得发送端的发送速率接近队列出口带宽的1/N,从而达到稳定状态(如图3所示);但是如果交点处于跳变段,或者是交点对应的队列长度大于丢弃上限,说明当前的丢弃概率无法阻止队列长度的增加,则RED队列无法通过当前的丢弃参数使得队列达到稳定状态(如图4所示).这就是目前RED算法在针对某种特性聚集流会发生调节失效,从而导致队列长度剧烈振荡的原因.

2.2 基于平衡点的自适应RED算法

根据以上对于RED算法的分析可知,一组固定参数配置的RED队列只能对参数满足特定要求的TCP流聚集产生很好的调节作用,而其中最重要的流聚集参数就是分组长度和流数目的乘积.由于一般情况下,互联网业务中流量大的业务的分组长度都较长(如FTP),为了简化模型,我们假设分组平均长度为1Kbyte,并且流聚集

中个每个 TCP 流的 RTT 时间相同,则对 G 函数起主要作用的就是流聚集中的 TCP 流的数目.

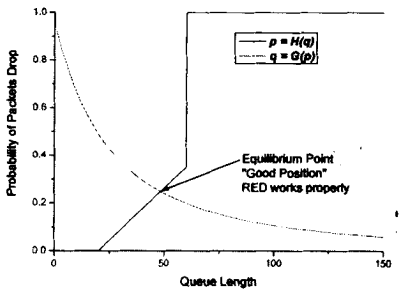

图 3 平衡点(RED 参数配置合理)

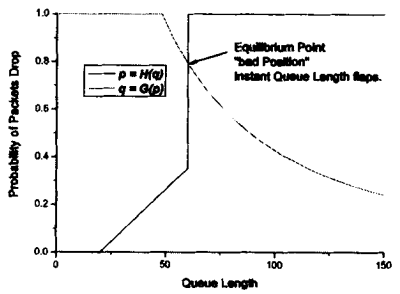

图 4 RED 算法失效时平衡点的位置

我们的基于平衡点的自适应 RED 算法是:通过求解方程,得到当前流聚集的平衡点,通过比较当前平衡点对应的平均队列长度和RED算法配置的丢弃上下限,来确定是否需要调整RED算法的最大丢弃概率.我们将平衡点对应的平均队列长度设为 Qe,平衡点对应的随机丢弃概率为 Pe,则当 Qe 小于 $\min_{th}+(\max_{th}-\min_{th})/4$ 时,说明当前的丢弃概率过大,我们需要将 $P\max$ 降低;当 Qe 大于 $\max_{th}$ 时,说明当前的最大丢弃概率已经无法对流聚集产生有效的调节,我们需要将 $P\max$ 提高;否则,说明当前的丢弃概率可以很好的调节流聚集,参数不需要调节.算法如下:

```
when RTT timeout
{
    Estimate Num_of_Flows;              // 估算流聚集内的 TCP 流数目
    Calculate Qe,Pe;                    // 计算平衡点对应的队列长度和丢弃概率
    if ( 0 <= Qe <= minth + ( maxth – minth )/4 )
        Pmax = Pe * X;          //X>1
    Else if ( Qe > maxth )
        Pmax = Pe * Y;      //Y>1
    Else
        No need to modify Pmax;
}
```

2.3 算法描述

通过上述算法描述,我们发现对 $P\max$ 的调整要基于 Qe 的计算结果,而流聚集中的流数目 N 是对 Qe 产生主要影响的参数,因此对于 N 的估计是该算法的另外一个重点.我们是通过相邻的 RTT 时间里入队的分组数目和丢弃的分组数目来估计流的数目,而在 TCP 流量控制的不同阶段,其估计算法又不相同,因此需要先估计流聚集中大多数流的状态.

2.3.1 对流聚集状态的估计

TCP 流的控制状态包括慢启动和拥塞避免.在慢启动状态时,每经过一个 RTT 时间,发送端的发送窗口将提高一倍,即发送端发出的分组数目是上一个 RTT 时间发送分组数目的两倍,而处于拥塞避免阶段的 TCP 发送端在某一个 RTT 时间内发送的分组数目只比上一个 RTT 时间内发送的分组数目多一个分组.因此,我们可以通过比较两个连续的 RTT 时间内到达队列的分组数目的变化,来确定流聚集中大多数流的控制状态.设连续两个 RTT 时间内进入队列和被丢弃的分组数目分别是:$enque_{pre}$,$drop_{pre}$,$enque_{cur}$ 和 $drop_{cur}$.设比值 State 为(参见公式(8)):

$$State = \frac{enque_{cur} + drop_{cur}}{enque_{pre} + drop_{pre}} \tag{8}$$

如果该比值大于 1.5,说明有超过半数流处于慢启动状态,则将这个流聚集按照慢启动状态进行流数目估计,如果该比值小于 1.5,则说明大多数流处于拥塞避免阶段,则按拥塞避免状态进行估计.

2.3.2 对流聚集中流数目的估计

2.3.2.1 慢启动状态

处于慢启动状态的 TCP 流,如果在前一个 RTT 时间内发送了 w 个分组,并且没有发生分组丢失,则在下一个 RTT 时间里会发送 $2w$ 个分组;如果在前一个 RTT 时间内发生了分组丢失,则在下一个 RTT 时间里会发送 $w/2$ 个分组,假设在流聚集中有 N 个流,并且丢弃是公平的,那么发生 $drop_{pre}$ 个丢弃分组属于 $drop_{pre}$ 个流,那么有(参见公式(9)):

$$\begin{cases} w \times N = enque_{pre} + drop_{pre} \\ 2w \times (N - drop_{pre}) + \dfrac{1}{2} w \times drop_{pre} = enque_{cur} + drop_{cur} \end{cases} \tag{9}$$

则

$$N = \frac{3(enque_{pre} + drop_{pre})}{2(2(enque_{pre} + drop_{pre}) - (enque_{cur} + drop_{cur}))} \tag{10}$$

2.3.2.2 拥塞避免状态

处于拥塞避免阶段的 TCP 流,如果在前一个 RTT 时间内发送了 w 个分组,并且没有发生分组丢失,则在下一个 RTT 时间里会发送 $w+1$ 个分组;如果在前一个 RTT 时间内发生了分组丢失,则在下一个 RTT 时间里会发送 $w/2$ 个分组,假设在流聚集中有 N 个流,并且丢弃是公平的,那么发生 $drop_{pre}$ 个丢弃分组属于 $drop_{pre}$ 个流,那么有(参见公式(11)):

$$\begin{cases} w \times N = enque_{pre} + drop_{pre} \\ (w+1) \times (N - drop_{pre}) + \dfrac{1}{2} w \times drop_{pre} = enque_{cur} + drop_{cur} \end{cases} \tag{11}$$

则

$$N = \frac{\sqrt{(enque_{cur} + drop_{cur} - enque_{pre})^2 + 2drop_{pre}(enque_{pre} + drop_{pre})} + (enque_{cur} + drop_{cur} - enque_{pre})}{2} \tag{12}$$

2.3.3 Pe 和 Qe 的求解

通过上述办法求得流聚集内 TCP 流的数目后,我们可以通过方程得到

$$\beta^2 P^3 + 2\beta\gamma P^2 + \gamma^2 P - \alpha^2 = 0 \tag{13}$$

其中 $\alpha = \sqrt{\frac{3}{2}} \cdot N \cdot S, \beta = (q_{max} - q_{min})/P_{max}, \gamma = q_{min} + B \cdot T_0$.

根据卡当公式,求解该方程的实数解,可以得到 Pe 和 Qe.得到 Pe 和 Qe 后,我们就可以依照在算法描述中策略,根据 Qe 和丢弃上下限的关系来调整 Pmax.

2.4 算法复杂性分析

基于平衡点的 RED 算法和传统的 RED 算法相比,增加了对分组转发情况的统计(到达分组数、入队分组数、尾丢弃分组数和随机丢弃分组数)、平衡点的计算和最大丢弃概率的修改过程.其中对分组转发情况的统计和最大丢弃概率的修改,与流聚集中的流的数量无直接关系,而平衡点的计算虽然与 TCP 流数目相关,由方程(13)和卡当公式可知,Pe 和 Qe 的计算复杂性为 $o(N)=1$.因此,基于平衡点的自适应 RED 算法的计算复杂性为 $o(N)=1$,即与通过队列的 TCP 流数目无关,因此该算法可以应用于高性能的核心路由器的队列管理中,具有较好的可扩展性.

3 实验和仿真

3.1 与RED算法的对比仿真实验

我们将上述设计的算法在NS2[9]上进行了仿真,仿真的环境包括5个发送客户端,两个路由器和一个接收客户端.连接情况为路由器 R1 连接 5 个发送客户端,连接带宽为 10Mbps,传输延迟为 5ms,路由器 R2 连接一个接收客户端,连接带宽为 50Mbps,传输延迟为 5ms,R1 和 R2 之间的连接上使用 RED 算法进行队列管理,连接带宽为 10Mbps,传输延迟为 10ms.在 5 个发送客户端上共有 160 个发送进程,业务类型为 FTP.

首先我们对传统 RED 算法进行仿真测试,然后对自适应 RED 算法进行测试.测试方案为依次使用 10,20,40,80,160 个发送进程,间隔 30s.RED 配置参数为 min_{th}=20,max_{th}=60,q_{lim}=200,w=0.02,下面分别是 P_{max}=1/16,1/8,1/4 的 RED 算法和自适应 RED 算法的测试结果.

图 5~图 8 分别显示了这 4 次仿真实验的即时队列长度变化情况和平均队列长度变化情况,通过图像我们可以清楚地看到,当流聚集内的流数量比较少的时候,采用较低的丢弃概率就可以达到很好的调节效果,这时候如果使用较大的丢弃概率会导致一些不必要的分组丢弃;当流聚集内流的数量比较多的时候,采用小的丢弃概率就不能起到调节作用了,而且会导致即时队列长度剧烈抖动,这时候应该采用较大的丢弃概率.所以针对不同的流聚集的特性应该采用不同的丢弃概率,而传统的静态配置参数的 RED 算法无法实现动态调整的功能.而我们设计的基于平衡点的自适应 RED 算法可以根据流聚集内流的数量来动态改变最大丢弃概率:当流的数目比较小的时候,算法就采用小的最大丢弃概率,避免不必要的分组丢失;当流聚集中的流的数量比较大的时候,算法就采用大的最大丢弃概率,防止队列长度剧烈抖动,使得 TCP 流尽快进入稳定状态.

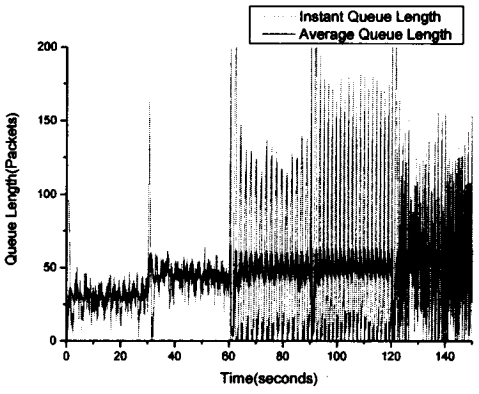

图 5 P_{max}=1/16

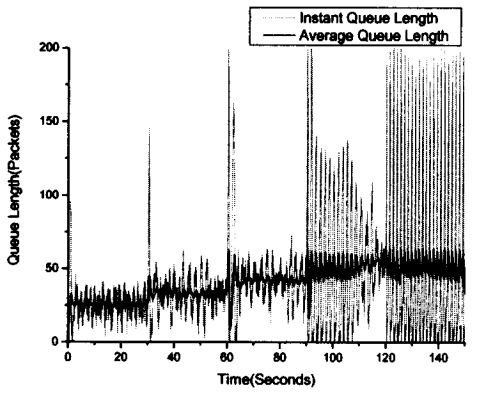

图 6 P_{max}=1/8

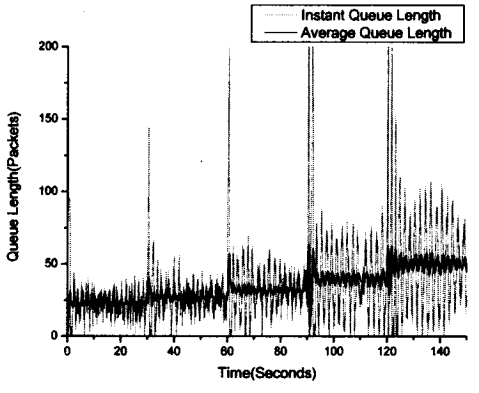

图 7 P_{max}=1/4

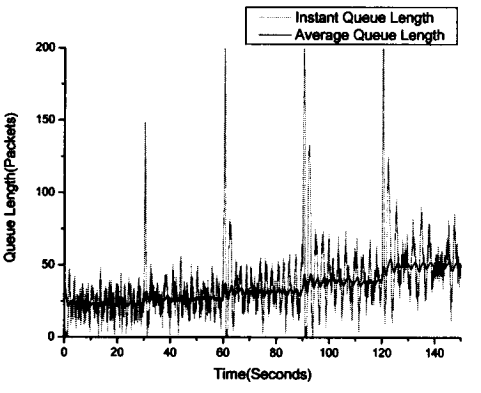

图 8 自适应 RED 算法

图9~图12显示的是四组仿真实验的统计参数对比,图9显示了4组实验的通过率的对比,四组仿真实验的通过率在小数据流的时候差别不大,在大数据流情况下,大丢弃概率(P_{max}=1/4)和自适应RED算法表现出来相对的优势.图10是四组实验的分组丢失率对比,结果和图8的意义一致.图10和图11将分组丢失率分为两个部分:即随机丢弃概率和尾丢弃率.图11显示大丢弃概率和自适应RED的随机丢弃特性比较稳定,能够随着流聚集中流数量的增大而提高丢弃概率,而小丢弃概率则没有这个能力.图12表明了自适应RED算法在流数量增加的时候依然可以保持较低的尾丢弃概率,而具有小丢弃概率的RED算法的尾丢弃概率却迅速增加.而尾丢弃很有可能是连续丢弃,这对TCP流的性能影响是显而易见的.

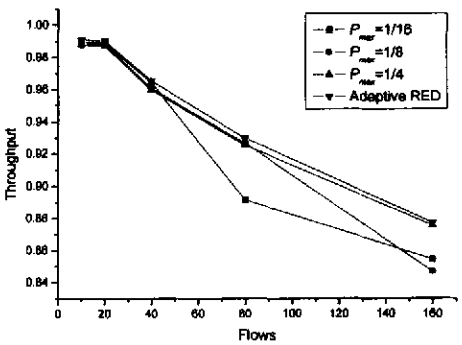

图9 通过率

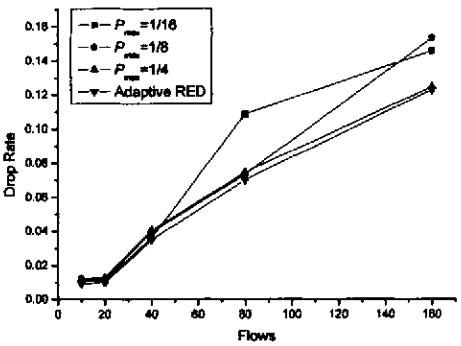

图10 分组丢失率

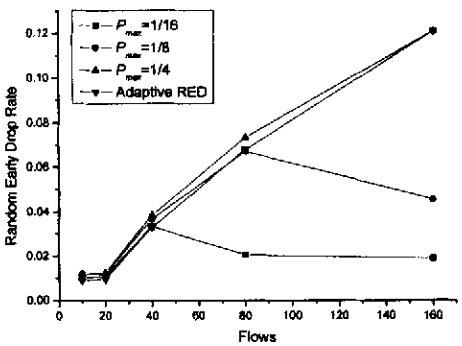

图11 随机丢弃率

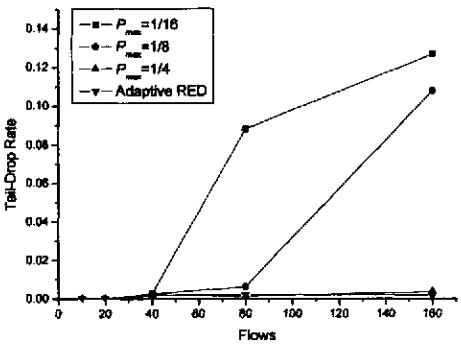

图12 尾丢弃率

3.2 与基于MIMD的RED改进算法的对比仿真实验

目前比较流行的自适应RED算法是由文献[3]提出的,该算法的核心是根据平均队列长度的变化来调整最大丢弃概率:如果当前的平均队列长度大于丢弃上限,也就是丢弃概率偏小,那么在上一次计算的基础上,将当前的最大丢弃概率乘以一个常数 $a(a>1)$;如果当前的平均队列长度小于丢弃下限,就将最大丢弃概率除以常数 $b(b>1)$,如果当前的平均队列长度处于丢弃上下限之间,就不需要调整最大丢弃概率.因此我们也将该算法简称为基于MIMD(Multiple Increased/Multiple Decreased)的自适应RED算法.而本文提出的基于平衡点的自适应算法直接计算平衡点,并根据平衡点来设置最大丢弃概率.在网络流量变化较小的时候,两种算法的结果差别不大;但是在网络流量变化剧烈的时候,本文提出的自适应RED算法能够比文献[3]中提出的基于MIMD的自适应算法更快收敛,达到稳定状态.

我们在NS系统上对两种自适应RED算法进行了两组对比实验.一组实验采用低突发率的TCP流:即先启动10个FTP流,持续20s后再启动10个FTP流,过20s后终止其中10个FTP流,再过20s结束实验.这样可以

测试两个算法在流量增加和减少的不同阶段的有效性.另外一组采用高突发的 TCP 流:即先启动 10 个 FTP 流,持续 20s 后再启动 80 个 FTP 流,过 20s 后终止其中 80 个 FTP 流,再过 20s 结束实验.

对比实验结果如图 13 至图 16.从图中我们可以清楚地看到:如果通过队列的 TCP 流量变化较小,两种算法的区别不大(MIMD 自适应算法的在 3 个阶段的平均队列长度变化为 28.611/39.911/ 28.286,收敛时间为 2s 左右;平衡点自适应算法的平均队列长度变化为 24.390/28.396/24.220,收敛时间小于 2s);当通过队列的 TCP 流数目剧烈变化的时候,本文提出的基于平衡点的自适应算法调整的结果具有更小的平均队列长度和收敛时间(MIMD 自适应算法的在 3 个阶段的平均队列长度变化为 28.661/45.094/29.456,收敛时间为 7s;平衡点自适应算法的平均队列长度变化为 24.390/42.156/24.847,收敛时间为 4s).而这两点对于服务质量控制算法是十分主要的,小的平均队列长度可以保证队列具有更小的平均排队时延,小的收敛时间可以保证该算法具有更高的可靠性.

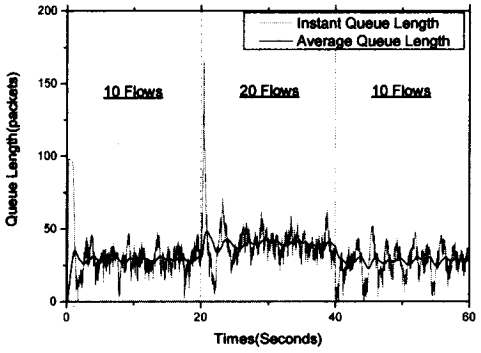

图 13 基于 MIMD 算法(低突发率状态下)

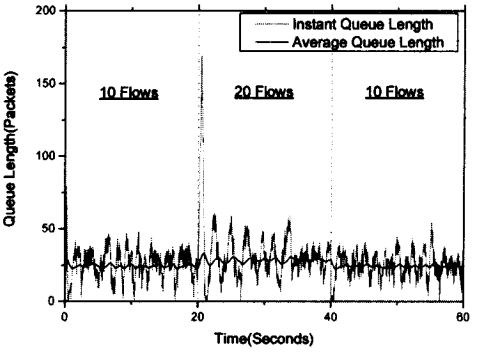

图 14 基于平衡点算法(低突发率状态下)

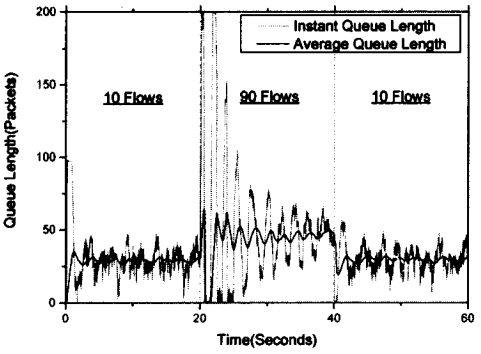

图 15 基于 MIMD 算法(高突发率状态下)

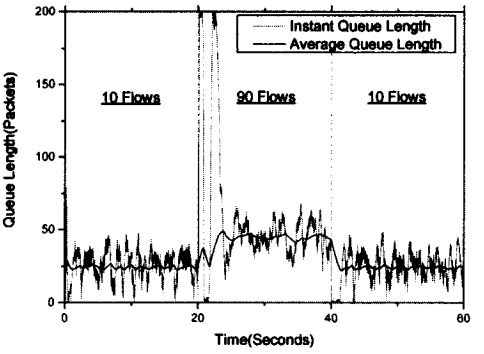

图 16 基于平衡点算法(高突发率状态下)

4 结论和相关工作

基于平衡点的自适应 RED 算法通过对进入队列的流聚集的特性参数进行统计,预测出流聚集中流的数量,并根据对平衡点的计算,动态地调整RED算法中的最大丢弃概率,使得基于平衡点的自适应RED算法具有更广泛的适应性.通过仿真实验更进一步证明了该算法的有效性.该自适应的 RED 算法针对的是流聚集,对流聚集内部的各个微流采取随机丢弃的原则,具有较好的公平性,因此该算法适用于面向流聚集进行服务质量控制的区分服务的队列管理,可以通过用户对端到端时延和分组丢失率的要求来确定 RED 队列相应的参数,从而保证用户的服务质量控制要求.同时,和基于 MIMD 改进 RED 算法的对比实验也说明了基于平衡点的自适应 RED 算法在队列长度收敛时间上具有一定的优势,可以使调整过程更短,更适用于高突发率的网络环境.

目前,RED队列管理是一个开放性问题,RED配置的其他参数也可以进行自适应调整,例如可以通过保持丢弃下限和丢弃函数的斜率不变,同时调整丢弃上限和最大丢弃概率,这样的调整适合于那些对时延不敏感而对丢失率敏感的业务.另外,计算平均队列长度时的权重 w 也可以进行调整,我们可以通过调整 w 来实现对突发业务的控制.总之,我们可以通过对 RED 队列管理算法中参数的不同调整策略来实现对服务质量参数的多种控制方法.

References:

[1] Floyd S, Jacobson V. Random early detection gateways for congestion avoidance. IEEE/ACM Trans. on Networking, 1997,1(4).

[2] Jacobson V, Karels MJ. Congestion avoidance and control. In: Proc. of the SIGCOMM'88. 1988.

[3] Feng WC, Kandlur D, Saha D, Shin K. A self-configuring RED gateway. In: Proc. of the Infocom'99. 1999.

[4] Floyd S, Gummadi R, Shenker S. Adaptive RED: An algorithm for increasing the robustness of RED's active queue management. 2001.

[5] Floyd S, Jacobson V. RFC2309 Recommendations on Queue Management and Congestion Avoidance in the Internet. 1998.

[6] Padhye J, Firoiu V, Towsley D, Kurose J. A stchastic model of TCP reno congestion avoidance and control. Technical Report, CMPSCI TR 99-02, Univ. of Massachusetts, Amherst, 1999.

[7] Stevens W. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC2001, 1997.

[8] Firoiu V, Borden M. A study of active queue management for congestion control. In: Proc. of the IEEE INFOCOM 2000. 2000.

[9] NS2. Network Simulator 2.0. http://www.isi.edu/nsnam