基于流的 MAWRED Linux 流量控制算法改进

The ameliorate algorithm of Linux transport controller which based on MAWRED of stream

王卫平　李 锐　何诗广　(中国科学技术大学 管理学院　230052)

摘要:Linux 在其内核中嵌入了的流量控制机制,但其在流量控制的队列调度算法方面仍然有所欠缺。本文在分析了 Linux 系统流量控制的内核实现基础上,提出了一种新的队列调度算法 – 基于流的 MAWRED 算法,并给出了其具体实现。

关键词:流量控制　队列调度　Linux　加权公平队列　随机早期检测　多平均队列加权随机早期检测

1　序言

Linxu 内核从 Kernel 2.1.105 开始支持 QoS(服务质量),其核心机理是在网络拥塞情况下,如何对不同数据进行调度和处理,即流量控制(TC)。如若要在 Linux 上实现 QoS,必须要确保在编译内核时选中 Kernel/User netlink socket,因为只有这样 TC 才能通过 netlink 与内核通讯。并要在编译时把队列规定和分类器都编进内核。应该说在目前通用的操作系统中,Linux 对流量控制的实现是做的最好的,但因为 Linux 系统本身的开源特性,它的队列调度策略却从 v2.2 起基本没有大的改进。因而本文将在对 Linux 的流量控制机理进行分析的基础上,对其队列调度策略进行进一步的完善。本文中的基于流的 MAWRED 流量控制算法为本文创新点。

2　Linux 流量控制原理

在 linux 中,对和网络相关的源代码都放在/Net 目录和/include/Net 目录下。首先我们了解一下 Linux 网络协议栈在没有 TC 模块时发送数据包的大致流程。当数据包进入系统核心时输入多路复用器将判断数据包的目的地址是否是本节点。如果是,这些数据包被送入网络传输层以等待进一步的处理,如果不是这些数据包会被送入转发区,同时本地节点的更高层应用程序产生的数据包也送入转发区内,系统查找路由表并确定这些数据包的下一跳地址。然后,这些数据包被排队送入输出队列中。

当数据包进入输出队列后,每个数据包的发送都会调用 dev_queue_xmit 函数(net/core/dev.c),在此判断是否需要向 AF_PACKET 协议支持体传递数据包内容,最后直接调用网卡驱动注册的发送函数把数据包发送给网卡。因而,如果要支持 QoS 功能,只要在 dev_queue_xmit 函数调用网卡驱动发送数据包之前,先调用流量控制的相关代码即可。有关代码如下:

```
int dev_queue_xmit(struct sk_buff *skb)
{
  ……
struct net_device *dev = skb->dev;
struct Qdisc *q;
……
  q = dev->qdisc;
  if (q->enqueue) {
    int ret = q->enqueue(skb, q);
    qdisc_run(dev);
    return;
  }
  if (dev->flags&IFF_UP) {
    ……
    if (netdev_nit)
      dev_queue_xmit_nit(skb,dev);
    if (dev->hard_start_xmit(skb, dev) == 0) {
      return 0;
```

```
        }
    }
    ..................
}
```

从上面的代码中可以看出，当 q - > enqueue 为假的时候，就不采用 TC 处理，而是直接发送这个数据包。如果为真，则对这个数据包进行 QoS 处理。

在了解过 Linux 的 QoS 入点后，我们再对 Linux QoS 的功能实现代码进行分析。Linux 进行流量控制的基本思路如下图 1 所示：

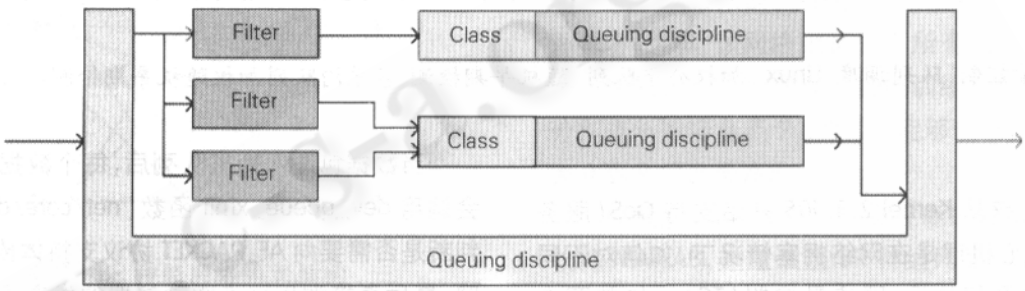

图 1　Linux 进行流量控制示意图

由图中可以看出，linux 的流量控制包含三个基本模块：filter（过滤器）、Class（类别）和 Queing discipline（排队策略）。当数据包进入内核的网络发送模块，由过滤器对数据包进行标记，送入不同的类别中，相应每个类别都有一个对应的队列策略负责数据包的最终输出调度。从 Linux 的流量控制整体架构中，可以看出队列策略是其核心所在。

队列策略的数据结构 Qdisc 在 include/net/pkt_sched.h 中定义：

```
struct Qdisc
{
struct Qdisc_head h;
int ( * enqueue) ( struct sk_buff * skb, struct Qdisc * dev) ;
struct sk_buff * ( * dequeue) ( struct Qdisc * dev) ;
unsigned flags;
struct Qdisc_ops * ops;/ * Qdisc 操作函数集合 * /
struct Qdisc * next;
u32 handle;
atomic_t refcnt;
struct sk_buff_head q;
struct device * dev;
  ......
char data[0];
};
```

在 Qdisc 中有一个指针型变量 ops 指向 Qdisc_ops 结构，在 Qdisc_ops 中定义了队列策略的各类动作。

最新版的 Linux 内核（v2.6.17）中实现了 14 种队列调度策略：atm、cbq、csz、dsmark、fifo、gred、hfsc、htb、netem、prio、red、sfq、tbf 和 teql。在 net\sched 目录下针对每种调度策略都有一个单独的 C 语言实现文件。每一个策略实现文件的结构大体相同，都是对 Qdisc_ops 结构中描述的队列操作函数在该策略情景下的具体实现。因而，如果要进一步改进 Linux 的流量调度策略，只要在 net\sched 目录下实现该策略的 C 语言实现文件，并在文件中提供 Qdisc_ops 中定义的接口函数即可。

3　加权公平队列（WFQ）

WFQ 是一个简单、动态的排队机制，它通过给重要的通信较高的优先级，以使它得到相对的更多的带宽，保证重要通信的服务。同时也兼顾低优先级流的公平，保证低优先级流的带宽占总带宽一定的比例，不至于发生“饥饿”的情况。其示意图如图 2。

WFQ 能够自动识别 IP 优先权。具体地说，WFQ 能够检测到被 IP 转发器标记了优先权的级别较高的数据包，并且通过为这些通信提供更快的响应时间，使这些数据包更快地传输。因此，随着优先权级别的增，

WFQ 在拥塞期间给这个会话分配了更多的带宽。WFQ 为每一个数据流分配一个权重。这个权重确定了排队等候的数据包的发送顺序。在这个方案中，权重小的通信可以先得到服务。

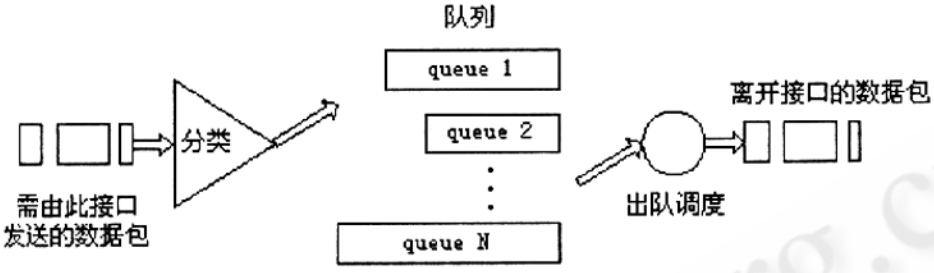

图 2　加权公平队列示意图

WFQ 对报文按流进行分类（相同源 IP 地址，目的 IP 地址，源端口号，目的端口号，协议号，TOS 的报文属于同一个流），每一个流被分配到一个队列，该过程称为散列，采用 HASH 算法来自动完成，并尽量将不同的流分入不同的队列。WFQ 的队列数目 N 可以配置。在出队的时候，WFQ 按流的优先级来分配每个流应占有出口的带宽。优先级的数值越小，所得的带宽越少。优先级的数值越大，所得的带宽越多。比如优先权字段的值为 7 的通信所得到的权重要小于优先权字段的值为 3 的通信。这样就保证了相同优先级业务之间的公平，体现了不同优先级业务之间的权值。为了决定为每一个队列分配的带宽，可以用单个数据流的字节总数来除所有数据流的总字节数。例如：如果对应于每一种优先权等级都有一个数据流，则每一个数据流将会得到的带宽是它的优先权级别 + 1 比上这个链接的总带宽。

例如：接口中当前有 8 个流，它们的优先级分别为 0、1、2、3、4、5、6、7。则带宽的总配额将是所有（流的优先级 + 1）之和，即：1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36

每个流所占带宽比例为：（自己的优先级数 + 1）/（所有（流的优先级 + 1）之和）。即，每个流可得的带宽比例分别为：1/36、2/36、3/36、4/36、5/36、6/36、7/36、8/36。

但是，如果对应于优先权级别 1 共有 18 个数据流，对应于其他的优先权级别各有一个数据流，则总数是：1 + 2（18）+ 3 + 4 + 5 + 6 + 7 + 8 = 70。优先权级别为 0 的通信将会得到总带宽的 1/70，而每一个优先权级别为 1 的数据流将会得到总带宽的 2/70。

又如：当前共 4 个流，3 个流的优先级为 4，1 个流的优先级为 5，则带宽的总配额将是：（4 + 1）* 3 + （5 + 1）= 21。那么，3 个优先级为 4 的流获得的带宽比例均为 5/21，优先级为 5 的流获得的带宽比例为 6/21。

由此可见，WFQ 在保证公平的基础上对不同优先级的业务体现权值，而权值依赖于 IP 报文头中所携带的 IP 优先级。当添加或者结束数据流的时候，实际所分配的带宽会连续发生变化。因此，通过分配每一个通信所获得的带宽，WFQ 可以适应于不断变化的网络环境。

4　MAWRED 算法

MAWRED（Multiple Average Weigthed Random Early Detction）多平均队列加权随机早期检测算法是 RED 算法的一种变体。RED 算法采用滑动窗口指数加权计算平均队长 Qavg，将平均队长与最小丢弃阈值 Tmin 和最大丢弃阈值 Tmax. 比较。当 Qavg < Tmin 时，不丢弃任何数据包；当 Tmin = < Qavg < Tmax. 时，以随机的概率丢弃到达的数据包，丢包概率随平均队长的变化在 0 和最大丢包概率 Pmax 之间变化；当 Tmax = < Qavg 时，丢弃所有到达的数据包。

MAWRED 对 RED 的改进体现在平均队列长度的计算上。RED 将所有数据放在一个队列中计算 Qavg，而 MAWRED 将流量进行分类，分别计算不同类别数据包的平均队长。在计算各类别数据包的平均队长时，高丢弃优先级的平均队长用它的队长和较低丢弃优先级的队长之和计算。即在计算较低丢弃优先级数据包的平均队长时，仅用较低丢弃优先级数据包的总队长计算；在计算较高丢弃优先级数据包的平均队长时，用较低丢弃优先级数据包和较高丢弃优先级数据包的总队长之和计算。MAWRED 算法描述如下：

假设有数据包类别 N_1、N_2、……N_s，其丢弃优先级 $L_1 < L_2 < \cdots\cdots < L_s$，队列长度为 Q_1、Q_2、……Q_s

当有数据包到来，

if 数据包类别为 n

计算类别 n 的平均队列长度 Qavgn = avg(Q_n + Q_{n-1} + …… + Q_1)

if Tminn = < Qavgn < Tmaxn

计算类别 n 的丢弃概率 P_n

以概率 P_n 丢弃到达的数据包

if Qavgn > = Tmaxn

丢弃所有达到的数据包

5 基于流的 MAWRED 流量控制算法

5.1 基于流的 MAWRED 流量控制算法思路

RED 技术对于那些对拥塞控制敏感的流(如 TCP 流)有用,在 TCP 流与非 TCP 流混合情况下并不理想,为了提供更有效的公平保障,满足流量服务区分的需要,可将 MAWRED 和 WFQ 配合使用,这样就可以实现基于流的 MAWRED。这是因为,在进行分类的时候,不同的流有自己的队列,对于流量小的流,由于其队列长度总是比较小,所以丢弃的概率将比较小。而流量大的流将会有较大的队列长度,从而丢弃较多的报文,保护了流量较小的流的利益。即使 WRED 和其他的队列机制配合使用,对于流量小的流,由于其报文的个数较少,所以从统计概率来说,被丢弃的概率也会较小,也可以保护流量较小的流的利益。

基于流的 MAWRED 示意如图 3 所示。

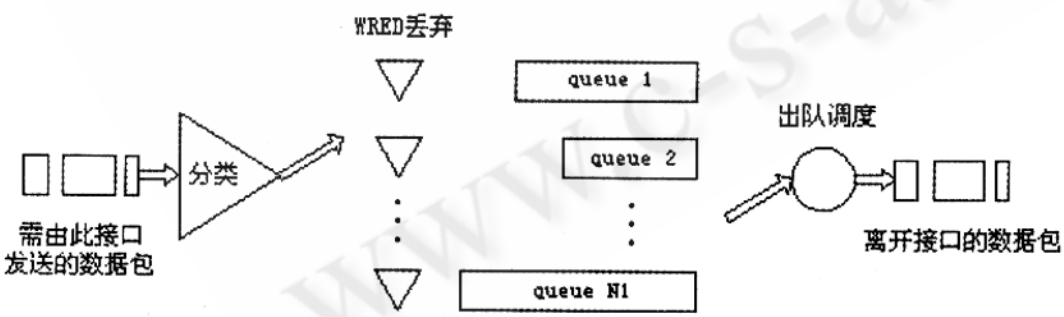

图 3　基于流的 MAWRED 示意图

在这种模式下,先利用 WFQ 可基于源和目的网络地址或者 MAC 地址、协议类型、源和目的端口以及帧中继数据链接标识符和服务类型等数据包报头参数动态地把通信划分成不同的数据流能力,将网络按流分类。而后利用 MAWRED 算法保证带宽在各个会话之间公平分享,保证数据量少的通信能够以一种及时的方式进行传输。这样以来,对于经过 WFQ 散列的每类队列,当出现网络拥塞时,将对每类队列采取 WRED 的报文丢弃策略,在具体实现中还可由用户设定队列长度的上限和下限。当队列长度低于下限时,不丢弃报文;当队列长度在上限和下限之间时,WRED 开始随机丢弃报文(队列长度越长,丢弃的概率越高);当队列长度高于上限时,丢弃所有报文。

5.2 基于流的 MAWRED 流量控制算法实现

我们在具体编程时考虑数据包的收发都是在系统运行时动态进行的,因此采用堆排序技术都数据包进行处理。堆排序是堆数据结构为队列向量向量的排序提供的一种有效技术。其基本思想是首先把队列构成堆,然后将堆顶元素(队列向量的最小值)与队列向量的最后一个元素交换,同时令堆的大小减少一个。下面给出我们的基于流的 MAWRED 算法的实现。

(1) 算法实现的主要步骤

每个数据包都会按照各自的带宽速率进入输出缓冲区,每个分组都有一个调度结束时刻,将这个时刻定义为每个分组的 F(i),i 代表第 i 个分组。具体算法如下:

· 使用分类器给分组分类,分类依据为数据包头的上文各项参数,并建立初始堆。

· 在调度模块中,每次都选取每个队列的第一个分组的调度时刻,采用最小堆排序的方法对其进行选择,选取最小 F(i)的分组进行调度。

· 重构堆。若所调度的分组所在的队列仍不为空,则向堆中插入该队列的下一个分组,直到队列为空。每次队列从空变成为非空时,都需要向堆中插入该队列的第一个分组。

(2) 主要函数

基于以上算法思路,给出我们的算法实现函数:

· fmawred_enqueue :把包插入该 Qdisc 所维护的队列中,如果该 Qdisc 有类,则先将包分类,然后再调用该类的 Qdisc 的 enqueue 函数进行下一步的入队

操作。

• fmawred_dequeue：从队列中取出一个可以发出的包。

• fmawred_requeue：把包重新放回原来的位置，这通常是在调用 dequeue 之后，包发往相应的设备时返回错误，需要把包重新放回队列中。

• fmawred_drop：从队列中丢弃一个包。

• fmawred_initQidc：初始化队列。

• initHeap：初始化堆栈。

• fmawred_reset：将 Qdisc 置为其初始状态，即清除所有的队列、停止所有定时器。

• fmawred_destroy：删除一个 Qdisc，除了描述它本身的数据结构以外，该 Qdisc 所有的类、过滤器均被清除。

• changeMaxp：修改最大丢弃概率。

• changMaxT：修改最大丢弃阈值。

• changeMinT：修改最小丢弃阈值。

• enHeap：数据包入堆。

• deHeap：数据包出堆。

• heapCounte：堆栈计数。

• fmawred_dump：返回用于诊断该 Qdisc 的数据。

基于流的 MAWRED 流量控制算法可以对最大丢弃阈值、最小丢弃阈值和最大丢弃概率 Maxp 进行逐流的调节，避免了 RED 机制下类似尾丢弃的现象。通过程序还证明了基于流的 MAWRED 流量控制算法的调度机制能够相对公平的调度各个队列的分组。而且在我们的实际编程中，每次都选取每个队列的第一个分组的 F(i) 进行比较，选出最小的进行调度。由于每次只需选出最小的，并不需要排序，考虑到时间复杂度，采用最小堆的结构，用一个有 64 个元素（对应 64 个类）的数组表示。它具有较低的时间复杂度 O(N10g2N)，并且由于堆排序的一个重要优点是它直接在队列向量数据上构造堆，所使用的附加空间仅仅是几个简单的临时变量，因而效率很高。

6 总结

在传统的 IP 网络中，网络设备对所有报文都无区别的等同对待，采用先入先出的调度策略进行处理，"尽最大的努力"将报文送到目的地，但对报文传送的可靠性和传送延迟不能提供任何保证。Linux 在其内核中集成进流量管理与控制的功能模块，提供很好的解决这一问题的结构，但 Linux 在流量管理的队列调度方面还不完善。本文提出这一新的队列调度算法很好的解决了队列调度的灵活性和高效性的两难问题，不仅能够完善 Linux 这方面的不足，也可应用与其它系统和网络设备中。

参考文献

1 《linux 防火墙内核中 Netfilter 和 Iptables 的分析》，刘建峰、潘军，微计算机信息，2006 年 03 期.

2 《基于区分服务的服务定制系统的实现》，彭丹、沈苏彬，南京邮电大学学报（自然科学版），2006.01.

3 《基于堆排序的 PQ + CBWFQ 路由器排队调度算法》，刘晏兵、孙世新，计算机工程，2006 年 01 期。

4 《linux 2.6 内核的内核对象机制分析》，丁晓波、桑楠，计算机应用，2005 年 01 期.

5 《IP QoS 主要体系结构及路由实现》，杨继峰，河南科技，2006 年 3 期.

6 《队列调度算法在网络中的应用研究》，杨永斌，计算机科学，2005 年 7 期.

7 《流量控制与 IP 服务质量》，黄启萍、唐亮贵，计算机工程，2006 年 11 期.

8 《支持区分服务的自适应队列调度算法》，刘辉、夏汉铸，计算机应用，2005 Vol. 25 No. 4.