Linux 环境下路由器中的网络带宽管理*

张焕强+, 吴志美

(中国科学院 软件研究所 多媒体通信和网络工程研究中心,北京 100080)

Traffic Control in Linux-Based Routers

ZHANG Huan-Qiang+, WU Zhi-Mei

(Multimedia Communication and Network Engineering Center, Institute of Software, The Chinese Academy of Sciences, Beijing 100080, China)

+ Corresponding author: Phn: +86-10-62645407, Fax: +86-10-62645410, E-mail: zhq@iscas.ac.cn, http://www.ios.ac.cn

Received 2003-03-27; Accepted 2003-09-09

Zhang HQ, Wu ZM. Traffic control in Linux-based routers. *Journal of Software*, 2005,16(3):462−471. DOI: 10.1360/jos160462

Abstract: Linux is widely adopted in routers nowadays, and traffic control is one of the most important functions for this kind of network-oriented operating systems. This paper studies the traffic control mechanism of Linux system and shows that the packet scheduling based traffic control mechanism adopted by current Linux kernel lacks a global view of system bandwidth, and is also short of efficient ingress scheduling. This can result in unnecessary CPU time wasting. To address this problem, a novel traffic control scheme is proposed, which is based on the CPU scheduling of the network protocol processing. By transplanting packet scheduling from the egress point of network interfaces to the soft interrupt handler that processes incoming packets, the new method can eliminate disadvantages of the old traffic control scheme without introducing any additional demerits. An implementation of the new traffic control scheme is given, and comparative experimental results show that the new mechanism has fewer overload than the old traffic control scheme in Linux, and can maintain the efficiency of traffic isolation and traffic policing while avoiding the CPU time wasting.

Key words: traffic control; bandwidth management; packet scheduling; CPU scheduling

摘 要: Linux 是目前广泛用于路由设备中的操作系统,而流量管理是这种网络操作系统的一个重要功能.研究了 Linux 系统的流量管理机制,发现当前 Linux 系统所采用的在网络接口的出口实现的基于网络包调度的流量管理机制缺乏对于网络带宽在系统范围的全局性的管理,也缺乏对于输入流的网络处理的有效管理和调度,从而造成不必要的 CPU 资源的浪费.为解决这一问题,提出了一种新颖的网络带宽管理机制,它通过调度网络协议处理所占用的 CPU 时间来实现带宽管理.这种新的机制将网络带宽的管理和调度从网络接口的出口点转移

* Supported by the National Grand Fundamental Research 973 Program of China under Grant No.G1998030407 (国家重点基础研究发展规划(973)); the Foundation of Beijing Science Committee of China under Grant No.H011710010123 (北京科委基金)

ZHANG Huan-Qiang was born in 1976. He is a researcher at Institute of Software, the Chinese Academy of Sciences. His current research areas are real-time system and multimedia communication. **WU Zhi-Mei** was born in 1942. He is a professor and doctoral supervisor at Institute of Software, the Chinese Academy of Sciences. His research areas are multimedia communication and broadband network.

到处理接收到的网络包的系统软件中断处理程序中,从而克服了原来的流量管理机制的缺点.通过系统实现和对比实验,发现该机制本身给系统带来的负载小于 Linux 原来的流量管理机制,同时可以提供更好的流量隔离,并且能够有效地节省用于网络处理的 CPU 资源.

关键词: 流量控制;带宽管理;包调度;CPU 调度

中图法分类号: TP393 **文献标识码**: A

Although more and more high-speed routers adopt ASICs to fulfill routing and packet forward, Linux-based routers still play important roles in Internet due to their scalability and flexibility. They are mostly suitable for the following arenas:

• Edge devices: In Diff-Serve architecture, the edge devices must carry out complicated and diversified processing to network packets, which results in a great need for scalability and flexibility. We consider all the following devices as edge devices: access routers, residential gateways, service gateways etc.

• Active network: The active network environment[1] need execute arbitrary codes in routers, thereby enabling the application-specific virtual networks. This requirement makes great demands on the scalability and programmable property of routers; so Linux-based router will be an appropriate decision for this scenario.

Traffic control is an indispensable function in most applications of Linux-based routers. Access routers need it to share network bandwidth between multiple organizations or even between different protocol families or different traffic types; residential gateways need it to fulfill QoS supporting for networked multimedia streams; and it's also a fundamental support in the implementation of a Diff-Serve enabled router.

This paper proposes a novel traffic control mechanism based on the CPU scheduling for network protocol processing. By transplanting packet scheduling from the egress point of network interfaces to the soft interrupt handler that processes incoming packets, the new mechanism can eliminate disadvantages of the old traffic control scheme without introducing any additional demerits.

The rest of the paper is organized as follows. Section 1 gives a description to Linux traffic control mechanism and its disadvantages. Section 2 shows the architecture of new traffic control scheme. Section 3 depicts the implementation related issues. Section 4 gives some experimental results. And finally comes a conclusion.

1 Traffic Control in Linux

Because its open source nature and extensive support to network, Linux is widely exploited in network devices. Linux offers a flexible traffic control architecture based on packet scheduling. Different queuing disciplines can be adopted for different purposes, varying from the simple classless queuing disciplines: FIFO, TBF, SFQ, RED to some complicated hierarchical queuing disciplines like CBQ (class-based queuing) and HTB (hierarchical token bucket).

1.1 Architecture of Linux traffic control

Linux implements traffic control through packet scheduling at the egress point of network interface. It bears a hierarchical architecture based on the complicated queue disciplines like CBQ or HTB. Figure 1 illustrates where packet scheduling takes place in Linux kernel and its composition[2].

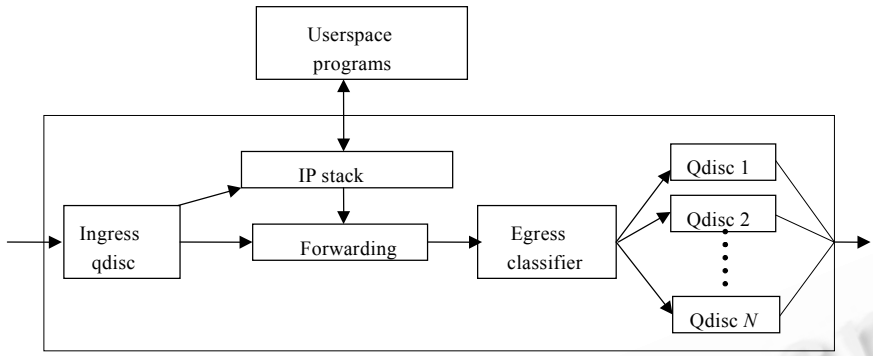


Fig.1 Architecture of Linux traffic control

The traffic control code in Linux kernel consists of the following components: queuing disciplines, classes, filters and policing. A queuing discipline is an algorithm that manages the queues of a device, either incoming (ingress) or outgoing (egress). It can be either classless queuing discipline which has no configurable internal subdivisions, or classified queuing discipline which contains multiple classes. Each of these classes contains a further queuing discipline, which may again be classified, but not necessarily. Filters fulfill the function of classification. A filter contains a number of conditions which if matched, make the filter match. The purpose of policing is to ensure that traffic does not exceed certain bounds, which can be accomplished by delaying or dropping packets. In Linux, because there is no ingress queue, policing can only drop a packet and not delay it.

1.2 The demerits of Linux packet scheduling

Though Linux traffic control architecture is flexible and powerful, it has the following disadvantages:

(1) It only allows classified egress traffic shaping. Though ingress queuing discipline can be applied to shape the incoming traffic before it enters the IP stack, but its function is very limited compared to powerful egress queuing disciplines.

(2) All queuing disciplines are attached to a certain network interface, so it can't hold a global view of the system, and global limitations to network traffic cannot be applied.

(3) CPU time wasting. As shown by Fig.1, egress queuing disciplines are implemented at the output point of a network interface and a packet has undergone the entire protocol processing and routing before it's scheduled at that point, so those packets discarded by queuing disciplines will incur unnecessary CPU time wasting (In a Pentium 133 PC, time taken by an IP packet processing is typically around 100μs), and excessive packets processing may incur live-lock[3]: a situation in which a router spends all its time on processing incoming packets that may be disposed instead of forwarding useful packets or servicing upper applications.

To address these limitations of Linux TC mechanism, the IMQ(Intermediate Queue) has been proposed[4], which helps address the first two limitations. By using dummy interfaces like imq0, imq1 etc., complicated egress queuing disciplines can be exploited for those incoming packets before they are queued to real egress network interfaces. This actually fulfills the function of ingress traffic control in an egress way. While the weakness of IMQ is obvious, a packet has undergone protocol processing before it's intercepted by a dummy interface. So the third problem with Linux TC is still there, even becomes worse.

2 Traffic Control

To address all these problems with Linux traffic control, we propose a novel traffic control scheme based on the CPU scheduling for network protocol processing. Instead of fulfilling traffic scheduling at the egress point of network interface, we transplant the packet scheduling into the network soft interrupt handler, where all packets are

scheduled before protocol processing.

2.1 Incoming packet processing in Linux

Linux kernel adopts soft interrupt mechanism to accomplish un-urgent tasks of an interrupt processing, which is called “bottom half” mechanism before version 2.4.0. The soft interrupt handlers are invoked from the return of interrupt handlers, and different from usual interrupt handlers. They can be intermitted by other urgent events.

The interrupt processing of incoming network packets follows the same pattern. First NIC (Network Interface Card) interrupts CPU once receiving packets from cable, the operating system then invokes the interrupt handler, which make the following processing: copying packets from NIC on-board memory to system memory, processing Ethernet header; queuing the packet into the receiving queue i.e. backlog queue. Then network soft interrupt handler will make other protocol processing and routing for the queued packets.

2.2 Architecture of network processing scheme

In 2.2.x Linux kernels, soft interrupt handler processes the incoming packets in a FIFO way and new 2.4.x kernel introduces trivial scheduling by giving different weight to traffics from different network interfaces, the new kernel promotes packet processing fairness among different interfaces.

Our goal is to add a more powerful and flexible scheduling mechanism to protocol processing in network soft interrupt handler, and create a system with a unified traffic control architecture, in which all incoming traffics are unanimously scheduled and policed before they undergo any further protocol processing, and eventually fulfill the function of traffic control through protocol processing scheduling.

Figure 2 illustrates the architecture of our scheduling framework. The framework is composed of the following parts: classifier, queues, scheduler, user interface module, and a user space tool.

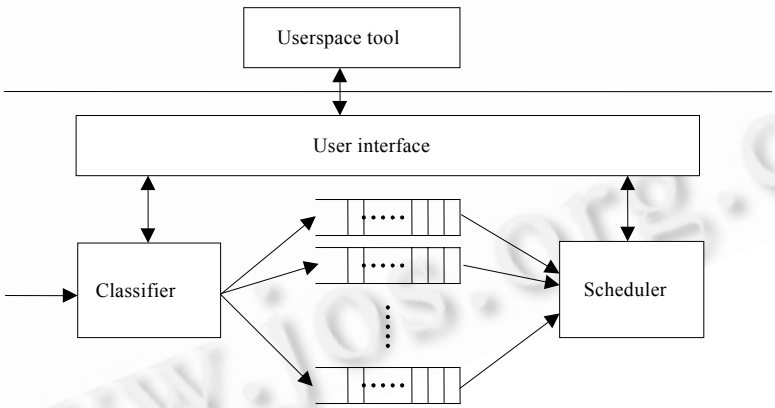


Fig.2 Architecture of new traffic control mechanism

The classifier classifies the incoming packets into different queues according to a certain criterion. This classifying criterion itself cannot take too much calculation and should be simple and easy to implement. The implementation codes should also be carefully considered in order to get a high efficiency. For IP packets, the classification can be based on IP addresses and TCP/UDP ports number, which can be easily drawn by simple indexing operations. Queues can be added or deleted on the fly. Each queue is identified by two properties: average bandwidth and peak bandwidth. The length of a queue should be in proportion to the bandwidth allocated to it. The scheduler plays the most important role in the whole architecture; it selects a packet from classified queues to make further processing. The next section will give a detailed description of the algorithm used by the scheduler. The user interface module and user space tool facilitate the queue managements and parameters setting of the scheduler.

3 Implementation of the Scheduling Scheme

For the sake of scalability and reduction alterations made to the Linux kernel, we only modify the code of the network soft interrupt handler in it, adding network processing scheduler register and unregister ability to the kernel. A network processing scheduler is identified by the structure illustrated in Fig.3; therefore the scheduler itself can be implemented in an independent Linux module, which implements all the needed functions of a network processing scheduler and registers to kernel in the module initialization function.

Our implementation allows multiple network processing schedulers to be registered into the system. Each scheduler owns a unique system ID and all the schedulers are linked together through a bidirectionally linked list (The next and prev pointers in the scheduler's data structure fulfill this-see Fig.3). A scheduler must implement two functions: skb_enqueue and skb_dequeue. The address of these functions should be passed on to the Linux kernel during the module initialization. The classifier is implemented in the skb_enqueue, and the scheduler is implemented in the skb_dequeue. All scheduler-specified data are stored in the struct sched_specified_data shown in Fig.3.

```
struct netbh_scheduler{
struct netbh_scheduler *next;
struct netbh_scheduler *prev;
unsigned int              id;
void (*skb_enqueue)( struct sk_buff *newsk);
struct sk_buff * (*skb_dequeue)(void);
struct sched_specified_data sched_data;
};
```

Fig.3 Data structure of a network processing scheduler

3.1 Implementation of classifier

To speed up the classification operation, fast cache and hash table facilities are adopted. Upon receiving a packet, a fast cache matching is first executed, which is accomplished by comparing ⟨Source IP, Source Port, Destination IP, Destination Port, Protocol⟩ of the packet header with cache content. If this fails, the hash value of incoming packet is calculated to launch a hash searching process. A successful cache match or hash search returns an entry of a classifying queue. Besides those specified flows, the system owns a default flow, which includes the rest flows passing through the system. When an incoming packet fails both cache matching and hash table searching, it will be classified into the system default flow.

Through this mechanism, the time consumed in packet classifying will be greatly decreased. This time can be kept fixed even when the number of flows is increased. Experiments show that in our platform which has a Pentium 133MHz CPU, this time is typically 2~6μs, with an average of 2.85μs.

When a flow is added to the system, a hash entry for that flow is created. The hash entry will be deleted when the flow is taken out from the system.

3.2 Implementation of the scheduler

Fig.4 illustrates the architecture of a network processing scheduler. It can be seen that the scheduler is composed of two components: a general scheduler and a bandwidth limiting scheduler. The general scheduler selects one qualified queue to be further scheduled by the bandwidth limiting scheduler that decides whether the first packet in this queue should be processed.

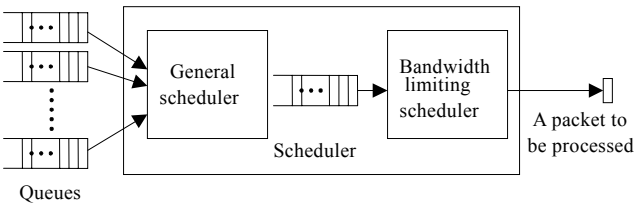


Fig.4 Architecture of the scheduler

The general scheduler can be any proportion-driven scheduler. Considering the small service time lag of the EEVDF[5] algorithm, we take it as the general scheduler in network processing scheduling.

As to the bandwidth limiting scheduler, traditional Linux traffic control schedulers can be considered. Among them, CBQ[6] adopts idle time calculation to estimate the actual rate of a flow, and fulfills the traffic regulation based on this information. Although HTB[7] does not resort to idle time calculations to do the traffic shaping, it do adopt the classic Token Bucket Filter. This leads to a better shaping result. So here in our network processing scheduler, we implement TBF (token bucket filter) algorithm and take it as the bandwidth limiting scheduler.

The detailed principles of the token bucket algorithm will not be recounted here. The interested readers can refer to Ref.[8]. It's difficult to implement the token bucket algorithm by its definition because it's hard to achieve an accurate fine-grained periodic timing in Linux kernel. So in the implementation, the periodic token updating is abandoned and tokens only get updated when the network software interrupt handler is invoked. Considering the irregular invocation of the network software interrupt handler, the token bucket algorithm is implemented based on the comparison of the two time variables: $T_{Token}(t)$ and $T_{Packet}(t)$ which are defined in the following definition section. The TBF algorithm is implemented in the following steps:

Definitions.

r : rate of token bucket;

N : depth of token bucket;

S_{Token}: size of a token;

$T_{Token}(t)$: time taken to consume up all tokens in the bucket;

$T_{Packet}(t)$: time taken to process the current packet;

TBF executing steps:

- Increase tokens, if current time is t and the last time when the algorithm is invoked is $t-\Delta t$, then the current $T_{Token}(t)$ can be calculated as:

$$T_{Token}(t)=\min\{N\cdot S_{Token}/r, T_{Token}(t-\Delta t)+\Delta t\}$$

- Process the current packet if the following formula holds true, else exit the algorithm:

$$T_{Packet}(t)<T_{Token}(t)$$

- Reduce tokens as follows after the packet has been processed:

$$T_{Token}(t)=T_{Token}(t)-T_{Packet}(t)$$

Each time the bandwidth limiting scheduler is invoked and the above TBF steps will be executed to determine whether or not the current packet should be processed.

3.3 Implementation of user interface

The function of the user interface modules is to facilitate the management of network queues. It is composed of two components: a user interface kernel module and a user-space application. The kernel module registers a character device to the system and implements the queue management functions in its IOCTL interface. Thus, the user-space tool can fulfill the operations of the queue management and parameters setting through the IOCTL calls

to this character device.

4 Experiments

4.1 Overload of the packet processing scheduler

We have checked the overload introduced by our scheduler mechanism by measuring the network throughput. Figure 5 illustrates the network configuration of the test environment. Two Linux PCs are connected together through a PC-based Linux router with two Ethernet cards. We use "netperf"[9] to test the throughput of varying-sized TCP and UDP packets. The experiment is executed in three configurations: bare Linux, Linux with IMQ enabled TC (called IMQ later), and Linux with network processing scheduler (called network scheduler later). In the experiment, to test the throughput, full-rate bandwidth is allocated to network flows between PC1 and PC2, and the throughput statistics shown here are from this flow. Figure 6 shows the throughput test results for TCP stream and UDP packets respectively.

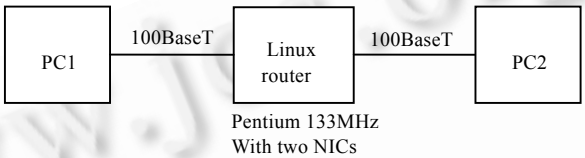
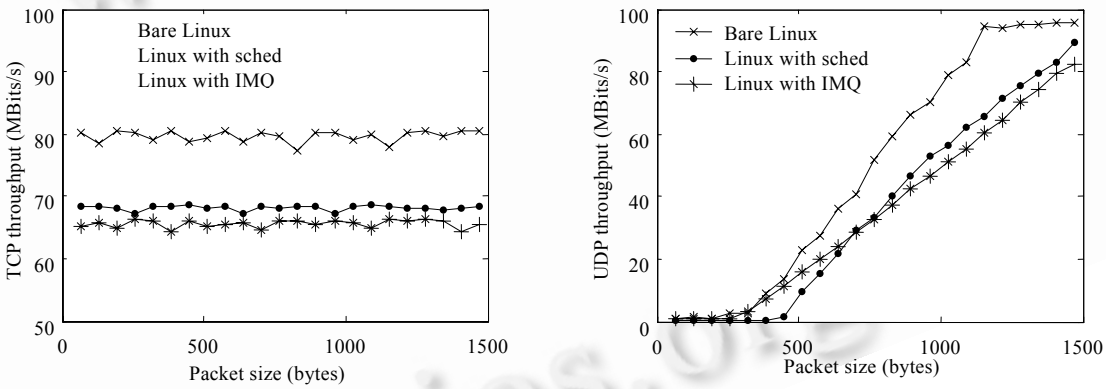


Fig.5 Throughput test environment



(a) Throughput of TCP packets (b) Throughput of UDP packets

Fig.6 Throughput statistics for different IP packet sizes

Figure 6(a) and (b) depict the throughput statistics of the IP packets with size ranging from 64 to 1500Bytes. The experimental results show that both the traffic control mechanisms introduce extra overload in network processing, while our solution has less overload than Linux TC tool. To give a quantitative result, CPU scheduling based traffic control mechanism has an average bandwidth of 68.1Mbits/s for TCP and 36.2Mbits/s for UDP, while TC tools has an average bandwidth of 65.6Mbits/s for TCP and 35.2Mbits/s for UDP.

4.2 CPU usage comparing

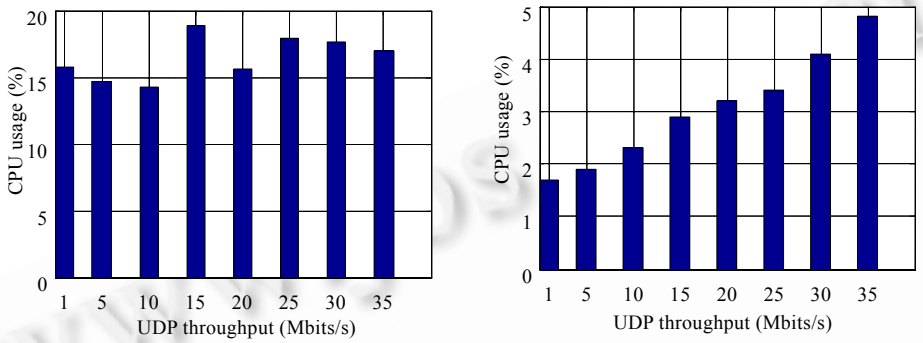
The CPU usage experiments are executed under the same network environment as shown in Fig.5. The CPU usage statistics of the Linux router are recorded during the UDP throughput tests from PC1 to PC2.

The data in Table 1 indicate the CPU usage statistics versus different UDP throughput. Here CPU usage means the CPU utilizing percentage of the Linux kernel, and different UDP throughputs are achieved by establishing a

flow in Linux router and allocating different network bandwidth to it. The flow establishing and bandwidth allocation are both done in IMQ case and network scheduler case, and the results of the two cases are compared in Table 1 and in Fig.7.

Table 1 CPU Usage statistics under different UDP throughput

Throughput (Mbits/s)	1	5	10	15	20	25	30	35
CPU usage (IMQ) (%)	15.8	14.7	14.3	18.8	15.6	17.9	17.6	16.9
CPU usage (Network Scheduler) (%)	1.7	1.9	2.3	2.9	3.2	3.4	4.1	4.8



(a) CPU usage of Linux with IMQ enabled TC (b) CPU usage of Linux with network scheduler

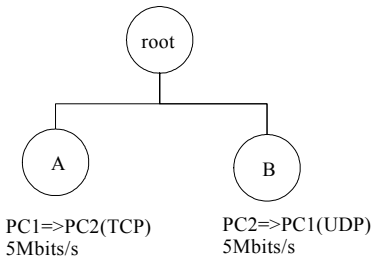
Fig.7 CPU Usage versus UDP throughput

It can be seen that when the bandwidth allocated to the UDP stream ranges from 1Mbits/s to 35Mbits/s, the CPU usage has no obvious trend in IMQ case, while it bares a distinct increasing trend in network scheduler case, increasing from 1.7% to 4.8%. The reason for this is IMQ does traffic policing in the IP layer, while those dropped packets have undergone network protocol processing before they get dropped, so the CPU usage keeps unchanged. Our network processing scheduler scheme avoids this demerit by adopting the packet policing in network software interrupt handler and the dropped packets are discarded before further processing are applied, thus the CPU time is saved.

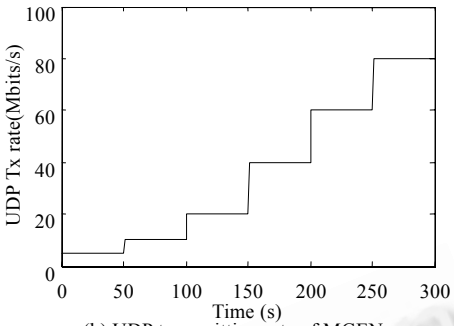
4.3 Traffic isolation effect

We also do experiments to substantiate the traffic limiting and traffic isolation effects of our new scheduling scheme. The experiments are still based on the network environment depicted in Fig.5. This time, two network flows A and B are explicitly created in the system (see Fig.8(a)), and they are all allocated 5Mbits/s bandwidth. Flow A is used for a TCP stream from PC1 to PC2, and flow B is used for a variable rate UDP stream from PC2 to PC1, and this UDP stream is generated by MGEN[10]. Our aim is to examine the influence of the UDP traffic on the TCP stream.

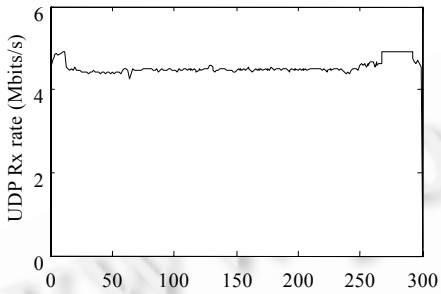
Figure 8 (b) depicts the UDP transmitting rate in PC2, which is not taken from measurements, but derived from MGEN transition script. Figure 8 (c) and (d) depict the UDP and TCP traffic rate in the IMQ case, and Fig.8 (e) and (f) illustrate the UDP and TCP traffic rate in the network processing scheduler case. In both cases, UDP transmitting rate, UDP receiving rate and TCP rate statistics are acquired during the same time period, i.e. they share the same *x*-axis.



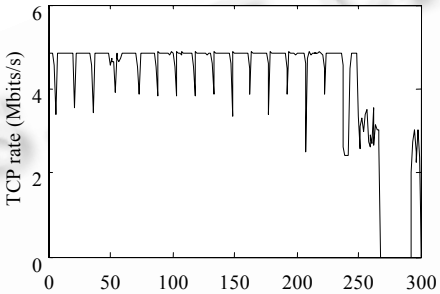
(a) Network flows in system



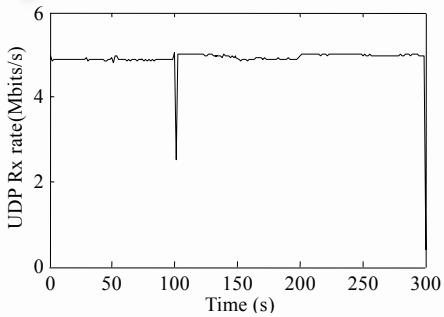
(b) UDP transmitting rate of MGEN



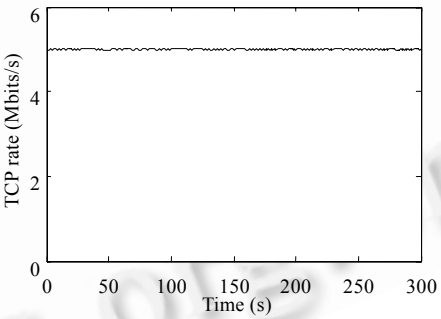
(c) UDP receiving rate in imq case



(d) TCP rate in imq case



(e) UDP receiving rate in network scheduler case



(f) TCP rate in network scheduler case

Fig.8 Traffic isolation test configuration and results

It can be seen from the test results that the traffic control based on the network processing scheduler has a better performance in traffic isolation. When UDP transmission rate reaches 80Mbits/s between 250s–300s, the TCP traffic rate in the IMQ case is terribly disturbed, while the network scheduler case can hold a steady result. Table 2 and Table 3 show some more detailed statistics.

Table 2 Traffic rate statistics for TCP stream

	IMQ	Network scheduler
Average TCP traffic rate (Mbits/s)	4.13816	4.9778
Standard deviation of TCP rate (Mbits/s)	1.4321	43.770

Table 3 Traffic rate standard deviation versus different time period

	0s−200s	200s−300s
IMQ (Kbits/s)	294.308	1.526
Network scheduler (Kbits/s)	52.018	9.344

Here the standard deviation in Tables 2 and 3 is not the deviation from the aiming rate (5Mbits/s), but a deviation value from its own average value which only reflects the fluctuation of the traffic rate.

5 Conclusion

This paper studies the traffic control mechanism of Linux system and proposes a novel traffic control scheme based on CPU scheduling of network protocol processing. By transplanting packet scheduling from the egress point of network interfaces to the soft interrupt handler, the new method can overcome the disadvantages of the old traffic control scheme. The implementation and experimental results show that the new traffic control mechanism has fewer overload than Linux TC, and can save packet processing CPU time and offer a better traffic control performance.

Though this traffic control mechanism is proposed and implemented in Linux system, considering the similarity of network sub-system of most UNIX systems, this mechanism can also be adopted in other networked operating systems.

References:

[1] Tennenhouse D, Smith J, Sincoskie W, Wetherall D, Minden G. A survey of active network research. IEEE Communications Magazine, 1997,(1):80−86.

[2] Almesberger W. Linux network traffic control — implementation overview. White Paper, April 23, 1999. http://lrcwww.epfl.ch/linux-diffserv/

[3] Mogul JC, Ramakrishnan KK. Eliminating receive livelock in an interrupt-driven kernel. ACM Trans. on Computer Systems, 1997,15(3):217−252.

[4] McHardy P. The intermediate queuing device. 2004. http://luxik.cdi.cz/~patrick/imq/

[5] Stoica I, Abdel-Wahab H. Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation. Technical Report 9522, Department of Computer Science, Old Dominion University, 1995.

[6] Floyd S, Jacobson V. Link-Sharing and resource management models for packet networks. IEEE/ACM Trans. on Networking, 1995,3(4).

[7] Martin Devera. 2004. http://luxik.cdi.cz/~devik/qos/htb/

[8] Partridge C. Gigabit Networking. Addison Wesley Publishers, 1994.

[9] Jones R, *et al*. Netperf – Software On-Line. 2004. http://www.cup.hp.com/netperf/NetperfPage.html

[10] Adamson B. The MGEN Toolset – Software On-Line. 2004. http://manimac.itd.nrl.navy.mil/MGEN/

[11] Qie XH, Bavier A, Peterson L, Karlin S. Scheduling computations on a software-based router. ACM SIGMETRICS Performance Evaluation Review, 2001,29(1):13−24.