LAWRRQ:A Queuing Management Algorithm for Scavenger Service

Xiaofeng Chen[1] Lingdi Ping[1] Xuezeng Pan[1]

[1]College of Computer Science, Zhejiang University, Hangzhou, Zhejiang 310027,P.R. China

Abstract

Scavenger Service is a non-elevated QoS technique proposed by the Internet2 project. This paper presents a LAWRRQ(Logarithmical Adaptive Weighted Round Robin Queuing) Algorithm for Scavenger Service. The Algorithm tunes proportion of scavenger service bandwidth logarithmically according to the number of active Scavenger Service flows to reach better performance and robustness. Simulations show that it provides more reliable bottom bandwidth guarantee for Scavenger Service flows while protecting Best Effort flows well.

Keywords: Queuing Management Algorithm, QoS, Logarithmic adaptation, Scavenger Service, DiffServ

1. Introduction

1.1. A brief introduction to Scavenger Service

Traditional IP network QoS researches focus on the way to improve the service of the 'important flows' and to guarantee that the packets of these flows get priority service over the packets of Best-Effort (BE) flows along the path. Many framework standards (such as IntServ[1] and DiffServ[2] from IETF) and protocols (such as RSVP[3] and MPLS[4]) have been raised to achieve QoS, but they are never widely employed ever since. That is because that the Internet today is enormous and to achieve QoS on the Internet and to give the 'important packets' better service, the devices along the flow path must recognize all these different standards and protocols. Even one device at the bottleneck fails to do this will make QoS unfeasible. On the basis of the IETF drafts[5,6], the Internet2 project[7] proposed a reverse-thinking QoS technology: QBone Scavenger Service [8]. In brief, it is a network mechanism that users (or their applications) voluntarily mark packets of some 'unimportant flows' with a specific DSCP (Differentiated Service Code Point) value (001000B). The routers implemented with Scavenger Service forward packets of BE flows with priority. And if there is some unused network capacity available, they forward packets of Scavenger Service (SS) flows. As we give the SS flows a downgrade service, this will increase the performance of BE flows. But the routers implemented with Scavenger Service don't strictly forward packets by priority. They still guarantee SS flows a minimal departure rate to avoid TCP connection time out and intolerable response delay of applications. All the unused network capacity can be used by SS flows if the whole capacity is more than BE flows requirement, so the resource won't be wasted. An outstanding advantage of Scavenger Service is that it doesn't need all routers' support along the flow path. Even some routers give same service to BE flows and SS flows, the other routers implemented with Scavenger Service function as throttle valves and the goal of Scavenger Service still can be realized. So, the Scavenger Service is a special deployable technology in the current Internet environment.

1.2. The Unresolved Problem

The Internet2 project deployed Scavenger Service on QBone, the QoS test bed of Internet2, and Abilene network. They also recommended several packets queuing mechanisms such as WRR, WFQ, MDDR [8]. References [8,9] presented some router configuration examples and references [9,10, 11] did some tests on Scavenger Service and gave the results. IETF has defined those Lower-Effort QoS technology like Scavenger Service as a new DiffServ PDB (Per-Domain Behavior) and published the relevant RFC document [12]. Because of the changing status of the network, the goal of packet queue scheduling algorithm for the Scavenger Service routers should be meeting the minimum demand of SS flows while affecting BE flows least on all occasions. How ever, the available queue management algorithms such as WRR, WFQ, MDRR [13] dealing with Scavenger Service are only able to allocate the minimum departure rate for Scavenger Service in static way

when BE flows are over subscribing network bandwidth. As the network capacity allocated for SS traffic in all is very low and the capacity for each SS flow is in inverse proportion to the number of SS flows, the SS flows may likely experience starvation if there are too many SS flows. However, allocating more capacity for Scavenger Service leads to resource misusing when there are few SS flows. It is necessary to develop new queue management algorithms aiming at the peculiarity of Scavenger Service.

2. Design Principle and Realization

The design principle of LAWRRQ queuing management algorithm aims at the peculiarity of Scavenger Service. It forwards BE packets with priority and allocates the spare capacity for SS flows. When BE traffic is heavy, it gives SS traffic a small guaranteed share of whole capacity. Furthermore, the algorithm adjusts the proportion allocated for SS traffic by tuning parameter according to the number of active SS flows while minimizes the impact on BE traffic.

2.1. LAWRRQ Queue Scheduling

LAWRRQ has two virtual queues: *vq_be* and *vq_ss*. The algorithm classifies arrived packets as BE packets and SS packets by the DSCP tag then enqueues them to the relevant virtual queue. When forwarding packets, the algorithm chooses one virtual queue by weighted round robin and dequeues the head packet of the queue to send.

To adjust the minimum guaranteed capacity for SS flows, the algorithm should first get the number of current active SS flows. So a Time Sliding Window (TSW) algorithm is included. It uses an array to record the arrival time of the last packet of each SS flow. A tag composed of the address and port of source and destination is used to classify a flow. Usually, a queuing management algorithm which maintains flow status table would not be practical by reason of the enormous cost brought by the huge status table when there are a great many flows. Whereas the LAWRRQ doesn't have such problem because of it's logarithmical characteristic. We can choose a rather small array with corresponding hash function and do no hash collision treating so as to limit the size of table. In this way, there might be some hash collisions and result in miscalculation of SS flow numbers. But the probability of hash collision is low with few SS flows. And when there are many flows, because of the characteristic of logarithmic functions: value rising very slowly when the independent variable goes up to a fairish value, the algorithm is insensitive with flow number error. As for the aim of LAWRRQ, a little error of flow number is tolerated.

The pseudo code of SS flows counting algorithm is as follows:

```
Upon each packet of SS flows arrival:
array[hash(flow_tag)]=current_time ;
if (current_time-last_time>=threshold_1){
    last_time=last_time+threshold_1;
    flow_count=0;
    for (i=0;i<=max_array_index;i++){
        if (current_time-array[i] <threshold_2)
            flow_count++;
    }
    tot_slices=access_table(flow_count) ; /*adjusts bandwidth
    allocation */
}
```

Two time thresholds *threshold_1* and *threshold_2* are used here. The algorithm calculates active SS flows each *threshold_1* instead of each SS packet arrival. This reduces the computing consumption remarkably. The chosen value of *threshold_2* should take it into account that packets of a flow having not appeared for a certain period of time doesn't mean the connection is disconnected. This may be as a result of packet dropping somewhere. Too small value of *threshold_2* may leads to *flow_count* vibrating. We make *threshold_1=0.1s* and *threshold_2=5s*, so the algorithm responds within 0.1 second after new SS flow appears and calculates out an old SS flow after it having disappeared for 5 seconds.

Having got *flow_count*, the number of SS flows, the algorithm would adjust the guaranteed minimum aggregate capacity for SS flows. LAWRRQ algorithm keeps the number of slices allocated for SS queue as 1 and adjusts the number of whole round robin slices. The proportion between SS flows count growth and SS bandwidth increment is the key of the algorithm. Increasing excessively will affect the BE flows and that increasing too little may plunge the SS flows in risk of starvation. LAWRRQ uses logarithmic function $f(x)=log_a(x)$ to fix on the proportion. Its going up curve is appropriate to the peculiarity of Scavenger Service. Different *a* makes different rising speed and suits to different network scale. Natural logarithm based on e, Euler's constant, is used to calculate in this paper.

Supposing $ln(flow\_count) \times tot\_slices = C$, total round robin slices amount is

$$tot\_slices = \frac{C}{ln(flow\_count)} \qquad (1)$$

In the equation, *C* is a constant（it can also be translated into the base *a*, then the numerator is 1）. It determines the actual proportion for Scavenger Service in total cycle with a certain SS flow count. Then, the share of BE flows and SS flows is

$$proportion\_vq\_be = \frac{tot\_slices - 1}{tot\_slices}$$

$$= \frac{\frac{C}{ln(flow\_count)} - 1}{\frac{C}{ln(flow\_count)}} = 1 - \frac{ln(flow\_count)}{C} \quad (2)$$

$$proportion\_vq\_ss = \frac{1}{tot\_slices} = \frac{ln(flow\_count)}{C} \quad (3)$$

It is distinctly impractical that the routers do complicated logarithmic computing when running. We should calculate the reciprocal of natural logarithms in advance and multiply them by the chosen C and then setup a table. The routers access the table to get *tot_slices* after calculating *flow_count* when running. A part of the table is shown as table 1.

Table 1.
SS flows number and corresponding total slices in a cycle

flow_count	*tot_slices*
1	150
2	100
3	63
4	50
5	43
6	38
7	35
8	33
9	31
10	30
……	……

We take *C=69.4* here to make *tot_slices=100* when *flow_count=2*. Because *ln(1)* is meaningless when *flow_count=1*, we should deal with the case designedly. Here we make *tot_slices=150* when *flow_count=1*. As a result of the characteristic of natural logarithm, when the value of *flow_count* goes up to a fairish value, *tot_slices* increases very slowly. So it is unnecessary to set many rows in this table. The function *access_table()* in the pseudo code above is the table looking up function.

LAWRRQ uses a slice turn counter *deq_turn* in dequeuing algorithm. When *deq_turn=0*, *vq_ss* is chosen. It can be seen that LAWRRQ allocates network capacity to BE and SS traffic by number of packets instead of actual bandwidth. As for the motivation of SS queuing management algorithm, it is more appropriate to give SS flows a bottom packet rate than a bit rate.

The dequeuing algorithm is as follows:

```
Upon interface free and queue not empty:
if (deq_turn_ >0) {
    if (length(vq_be)>0){
        dequeue vq_be;
        deq_turn++;
        if (deq_turn>=tot_slices);
            deq_turn = 0;  /*if counter increases to
                tot_slices, begin next cycle*/
    }
    else {
        dequeue vq_ss; /*if vq_be is empty，dequeue vq_ss*/
        deq_turn=1;
    }
}
else {
    deq_turn=1;
    if (length(vq_ss)>0)
        dequeue vq_ss;
    else
        dequeue vq_be;
}
```

2.2. Buffering Management of LAWRRQ

The two queues may use their own physical buffer space separately. But this evidently wastes buffer memory when there is only one kind of traffic busy. Another way is two virtual queues share one physical buffer space and the inequality *length(vq_be)+length(vq_ss) <BUFFER_SIZE* is used to test whether the buffer is full. But this test is not enough. We can see that BE flows bandwidth wobbling violently when SS flow number increases to a certain value in the simulations in part 3 of this paper. By queue tracing, we can find that the cause of the phenomenon is that packets of SS flows and BE flows are treated fairly in enqueuing but not in dequeuing. When dequeuing, because of the peculiarity of Scavenger Service, the SS packets are given much less opportunity than BE packets. Consequently the SS packets tend to 'pile-up' in the buffer. Upon that, we add algorithm to unilaterally protect BE flows as following. At the arrival of BE packets, if *length(vq_be)+length(vq_ss)<BUFFER_SIZE* is not true, then the algorithm checks whether the length of SS queue is greater than half of *BUFFER_SIZE*. If so, the algorithm drops the tail of *vq_ss* to vacate space then enqueues the arrived packet to the tail of *vq_be*.

The manage mechanism inside the virtual queues can be those in common use such as Drop-Tail, RED. In this paper, we use simple Drop-Tail. The pseudo code of enqueuing algorithm is as follows:

```
Upon each packet arrival:
if (packet is belong to SS flow){
    if (length(vq_be)+length(vq_ss)<BUFFER_SIZE)
        enqueue packet to vq_ss;
    else
        drop packet;
}
else{
    if (length(vq_be)+length(vq_ss)<BUFFER_SIZE)
        enqueue packet to vq_be
    else if (length(vq_ss)> BUFFER_SIZE/2){
        drop tail of vq_ss;
        enqueue packet to vq_be;
    }
```

else

drop packet;

}

3. Simulations and Analysis

3.1. Tools and Environment

NS Version 2.26[14] was used to do all the simulations below and we realized LAWRRQ algorithm by c++. The topology of simulated network is shown as Fig. 1.

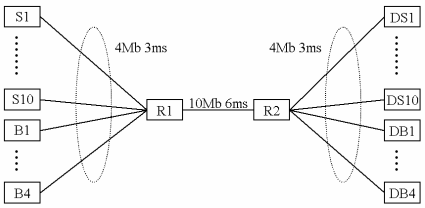

Fig. 1. The network topology

In Fig.1, R1 and R2 are routers connected by a bi-directional link with 10Mbps bandwidth and 6ms latency. The LAWRRQ queuing management algorithm is implemented in the interfaces of R1 and R2 connecting each other. The buffer is 128 packet-size. Except this connection, all other links are 4Mpbs of capacity and 3ms of latency with common FIFO Drop-Tail queue. S1 to S10 are 10 Scavenger Service data sources communicating respectively with DS1 to DS10. B1 to B4 are 4 Best-Effort data sources communicating respectively with DB1 to DB4. All communications take FTP as data source and use TCP/Reno protocol with TCP window size of 15 and packet size of 1000bytes.

3.2. Realization of Scavenger Service

To validate the realization of Scavenger Service, simulation scenario is made as following: BE source B1 starts at the beginning. At time of 2.0s, all 10 SS sources start one by one with 2s interval. At 22.0s, the left 3 BE sourced start one by one with 2s interval. The simulation ended at 30.0s. In this simulation, we observed the aggregate bandwidth of two kinds of traffic. Simulation result is shown in Fig. 2. The result shows that SS flows fully utilized the spare network capacity when BE traffic load was light. The increment of SS flow number didn't prevent the BE flow form reaching its utmost bandwidth. After B2 started, SS flows began to give up possession of network capacity voluntarily till it only held a tiny guaranteed capacity and the BE flows consumed most portion of the capacity. Scavenger Service was fulfilled well.

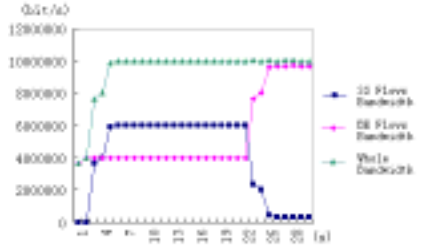

Fig. 2. Realization of Scavenger Service

3.3. Protection for BE Queuing

To observe protection for BE flows better, we reduced total buffer to 64 packet-size in this simulation. All 4 BE sources started at the beginning of simulation. After 5s, all the SS sources started one by one with 0.1s interval. The simulation ended at 25s. In this simulation, we focused on bandwidth and packet lost of one BE flow (B1 source here). First, we ran the simulation with buffer running out tested by length(vq_be)+length(vq_ss)<BUFFER_SIZE only. Result is shown in Fig.3.It can be seen that SS flows number increment caused intense BE packets dropping accompanied with BE bandwidth vibrating fiercely when BE traffic was heavy. As discussed in chapter 2.2, this phenomenon is due to that SS packets tend to heap up in the buffer. When there are many SS flows and TCP window of these flows expands, the virtual queue vq_ss would consume most of buffer and impact BE flows. Fig.4 gives the result of the same simulation after the BE flows protecting algorithm in chapter 2.2 was added.

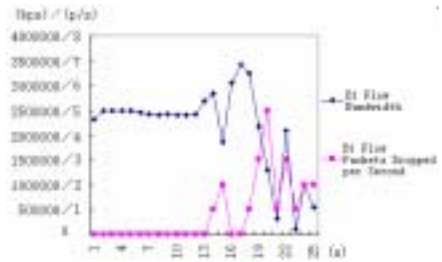

Fig. 3. Only test buffer full when enqueuing

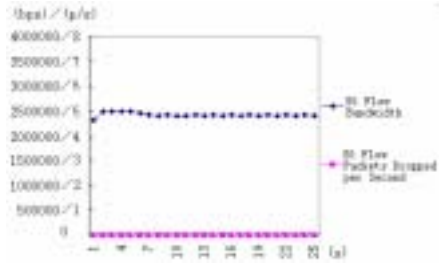

Fig. 4. BE flows protecting algorithm added

3.4. Reaction to SS Flows Number When BE Traffic is Over Subscribing

In this simulation, scenario was set as following. All 4 BE sources start at the beginning of simulation. At time of 2.0s, all 10 SS sources start one by one with 2s interval. Simulation runs for 5s more after last SS source S10 starts at 20.0s. We focused on aggregate bandwidth of SS flows in this simulation. To compare, we first chose one queuing management algorithm that configures the SS traffic minimum departure rate in static way to run the simulation.

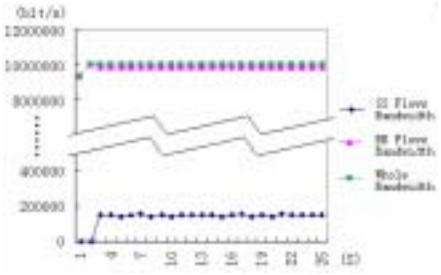

Fig. 5. The result of WRR

Fig. 5 is the result of WRR, which is one of the queuing disciplines recommended by Internet2. SS traffic minimum departure rate was set to 1.5% of link capacity, i.e. 150Kpbs. It can be seen that as the active SS flows number rose form 1 to 10, the aggregate bandwidth they attained remained constant.

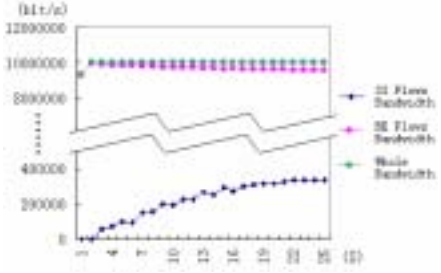

Fig. 6. The result of LAWRRQ

The result of LAWRRQ in Fig. 6 shows that the aggregate bandwidth attained by SS flows went up approximately as a logarithmic curve. This meets the design aim. Compared with WRR in Fig. 5, LAWRRQ adapts the aggregate minimum bandwidth of SS traffic to the number of active SS flows. When there are few SS flows, it allocates less guaranteed bandwidth to SS traffic in order to give the capacity to BE traffic at best. This will make BE flows perform better. When the number of active SS flows increases, it allocates more bandwidth to SS traffic so as to turn SS flows away from starvation. This gives SS flows more robustness.

4. Future Work

LAWRRQ in this paper and other queue principles proposed by Internet2 for Scavenger Service haven't taken the case that individual flows inside the aggregate flows of SS are heterogeneous into account. In fact, because SS traffic is given a very small portion of whole network capacity when BE traffic is heavy, the unfair distribution of bandwidth between individual flows inside SS traffic is much more critical. How to introduce algorithms with high performance and low cost to balance bandwidth allocation inside SS aggregate flow to Scavenger Service queuing management algorithms is the next issue.

5. References

[1] R. Braden, D. Clark, S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC1633, June 1994.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss. An Architecture for Differentiated Service. RFC2475, December 1998.

[3] J. Wroclawski. The Use of RSVP with IETF Integrated Services. RFC 2210, September 1997.

[4] F. Le Faucheur, L. Wu, B. Davie, S. Davari, P. Vaananen, R Krishnan, P. Cheval, J. Heinanen. Multi-Protocol Label Switching (MPLS) Support of Differentiated Services. RFC3270, May 2002.

[5] R. Bless and K. Wehrle. A Lower Than Best-Effort Per-Hop Behavior. http://quimby.gnus.org/internet-drafts/draft-bless-diffserv-lbe-phb-00.txt. Work in Progress, September 1999.

[6] B. Carpenter , K. Nichols. A Bulk Handling Per-Domain Behavior for Differentiated Services. http://www.ietf.org/proceedings/01aug/I-D/draft-ietf-diffserv-pdb-bh-02.txt. Work in Progress, January 2001.

[7] Internet2-Home. http://www.internet2.edu/. January, 2001.

[8] QBone Scavenger Service (QBSS). http://QBone.internet2.edu/qbss/.

[9] SLAC QBSS Testbed. http://www-iepm.slac.stanford.edu/monitoring/qbss/qbss.html.

[10] Juniper QBSS experiment. http://www.transpac.org/qbss-html/.

[11] Stanislav Shalunov. Testing QBSS config at Advanced.org: Good. http://archives.internet2.edu/guest/archives/i2ss-dt/log200105/msg00000.html

[12] R. Bless, K. Nichols, K. Wehrle. A Lower Effort Per-Domain Behavior (PDB) for Differentiated Services. RFC3662, December 2003.

[13] VEGESNA S. IP Quality of Service (Cisco Networking Fundamentals)[M]. Cisco Press. 23 January, 2001

[14] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/index.html.