Hindawi
Journal of Electrical and Computer Engineering
Volume 2018, Article ID 4576245, 9 pages
https://doi.org/10.1155/2018/4576245

Research Article

An Application-Level QoS Control Method Based on Local Bandwidth Scheduling

Yong Wang, Fu Xu, Zhibo Chen, Yu Sun, and Haiyan Zhang

School of Information Science and Technology, Beijing Forestry University, Beijing 100083, China

Correspondence should be addressed to Fu Xu; xufu@bjfu.edu.cn

Received 9 November 2017; Revised 23 April 2018; Accepted 10 May 2018; Published 6 June 2018

Academic Editor: Stefano Vitturi

Quality of service (QoS) is an important performance indicator for Web applications and bandwidth is a key factor affecting QoS. Current methods use network protocols or ports to schedule bandwidth, which require tedious manual configurations or modifications of the underlying network. Some applications use dynamic ports and the traditional port-based bandwidth control methods cannot deal with them. A new QoS control method based on local bandwidth scheduling is proposed, which can schedule bandwidth at application level in a user-transparent way and it does not require tedious manual configurations. Experimental results indicate that the new method can effectively improve the QoS for applications, and it can be easily integrated into current Web applications without the need to modify the underlying network.

1. Introduction

Improving bandwidth efficiency is a focus in academic research and industrial innovation. Internet adopts a best-effort service model, which lacks the capability of QoS guarantee inherently [1]. Although traditional methods such as Resource Reservation Protocol (RSVP) [2] and Differentiated Service (DiffServ) [3] can improve the QoS for Web applications effectively, they require modifying the underlying network [4]. For example, Neto proposed a multicast-aware RSVP for class-based networks [5] and FU Qi proposed a service-aware multipath QoS strategy, which achieved the fair use of different types of traffic channels and wireless links [6]. All these methods require modifying the underlying network. Allowing for the diversity of network types, these modifications are not always feasible [7, 8]. Some QoS methods such as Traffic Control (TC) [9] do not require modifying the underlying network, which schedule bandwidth based on network protocols or ports. Manual configurations are inevitable when using these methods. This process is very cumbersome and requires highly specialized expertise. Even network experts cannot handle this task easily. Furthermore, some applications such as uTorrent [10] use dynamic ports and traditional port-based bandwidth control methods cannot deal with them.

With the rapid emergence of Web applications, the demand for application-level QoS control on local computers is growing. The disorderly competition in bandwidth usage may cause bandwidth-sensitive applications working abnormally. Although the traditional fixed bandwidth allocation strategy can avoid this problem and guarantee the QoS, it can lead to bandwidth idling.

A new QoS control method is proposed in this research, which uses an improved token bucket algorithm and can schedule bandwidth at application level in a user-transparent manner. This method dynamically allocates bandwidth based on the bandwidth requirements of applications, which achieves a good balance between bandwidth usage efficiency and QoS guarantee. It can maximize the bandwidth usage while guaranteeing the QoS for applications running in the same computer. In addition, the new method does not require modifying the underlying network and can avoid the problem of tedious configurations.

2. Related Work

2.1. DiffServ. DiffServ is often used to ensure the QoS of backbone networks, which classifies data flows into different levels according to their QoS requirements. High-level data flows are preferentially transmitted than those at low levels

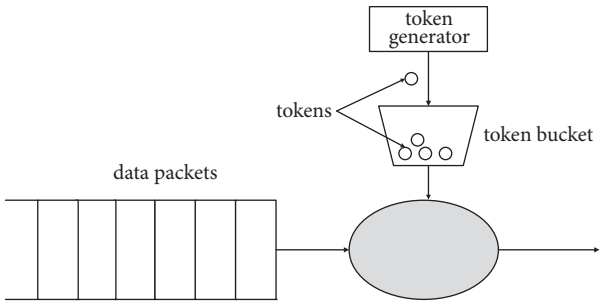

Figure 1: Flowchart of token bucket algorithm.

when encountering congestions. More details on DiffServ can be found in RFC2474 [11].

DiffServ has become the current mainstream QoS architecture because of its good scalability, simplicity, and operability. However, DiffServ is not an end-to-end method and it cannot be used without modifying the underlying network.

2.2. Token Bucket Algorithm. As a classical bandwidth control method, token bucket algorithm [12] has good capability in congestion processing, which consists of three components: token, token bucket, and token generator (Figure 1). Token controls the transmission of data packets and token generator generates tokens to fill into the token bucket. When there are enough tokens in the bucket, data packets can be transmitted. Otherwise, they will be buffered or discarded.

2.3. Review of Existing Methods for Bandwidth Control. There are lots of researches in QoS guarantee. Most of them schedule bandwidth at router or network layer, which are not application-level based scheduling methods, so that they cannot avoid the confliction of bandwidth usage in the local computer.

Lin N. proposed a QoS control method based on DiffServ and token bucket algorithm [13], which designed three services of QoS: Guaranteed Service (GS), Control Load Service (CLS), and Best Effort Service (BES). Protocols and ports in the packet headers were used to distinguish these services. This method can guarantee the QoS for applications with GS type. It cannot deal with the applications using dynamic ports and requires modifying the underlying network and making port-based manual configurations.

Kidambi proposed a token bucket based method, in which bandwidth was equally divided among data flows [14]. This method can guarantee the QoS for applications but has the problem of idle bandwidth due to the fixed bandwidth allocation strategy.

Cucinotta proposed a QoS control method for real-time applications, which can guarantee the QoS for real-time applications [15]. This method uses fixed bandwidth allocation strategy, which has the similar problem with Kidambi's method.

Hierarchical token bucket [16] method can avoid idle bandwidth, but it requires tedious manual configurations based on network protocols or ports. It is not an application-level method and cannot schedule the bandwidth for applications using dynamic ports.

3. The New Application-Level QoS Control Method

Bandwidth is a key performance indicator for many Web applications. For example, a VOIP application using G.711 codec requires at least 64 KB/s bandwidth to guarantee good call quality [17]. In the new method, priorities are assigned to applications according to their bandwidth requirements. An improved token bucket algorithm is designed, which can dynamically schedule bandwidth based on the priorities of applications. The following two steps describe the new method in detail.

3.1. Application-Level Bandwidth Scheduling. Popular operating systems (OS) such as Linux, Windows, and MacOS usually support some priority mechanisms. These mechanisms are more like a gentleman's agreement and many applications do not follow them strictly, so that these mechanisms cannot solve the problem in bandwidth control. Moreover, priorities assigned by OS are based on CPU timeslicing instead of bandwidth [18], so that they cannot guarantee the QoS for Web applications.

To achieve application-level bandwidth scheduling, the network packets should be associated with their corresponding processes. The hooking mechanism [19] provided by OS can be used to do this. For example, WFP (Windows Filtering Platform) [20] hooking mechanism can be used to filter network packets on Windows. Other operating systems such as Linux or MacOS also have similar mechanisms.

The new application-level bandwidth scheduling method consists of four steps. Let us take Windows as an example.

(1) Build a priority database to store the fingerprints of applications. The fingerprint can be the MD5 hash of an application, the application name, or other tags that can uniquely identify the application. In this research, MD5 hashes are used to generate the fingerprints. Except the fingerprints, the bandwidth requirements and application priorities are also stored in the priority database.

(2) Hook the packet sending and receiving functions in network protocol libraries and use WFP interfaces to obtain the corresponding process IDs (PIDs) for these packets. Retrieve the application's full paths through PIDs and generate fingerprints using these paths.

(3) Query the priority database using the fingerprints to retrieve the application's priorities. If an application is not configured in the priority database, it will be assigned the lowest priority.

(4) Schedule bandwidth at application level using the method described in Section 3.2.

3.2. Bandwidth Dynamic Scheduling Method. There are two typical bandwidth scheduling methods. One method is to let applications freely compete for bandwidth, which can maximize bandwidth usage and has no problem of idle bandwidth. The drawback is that it cannot guarantee the

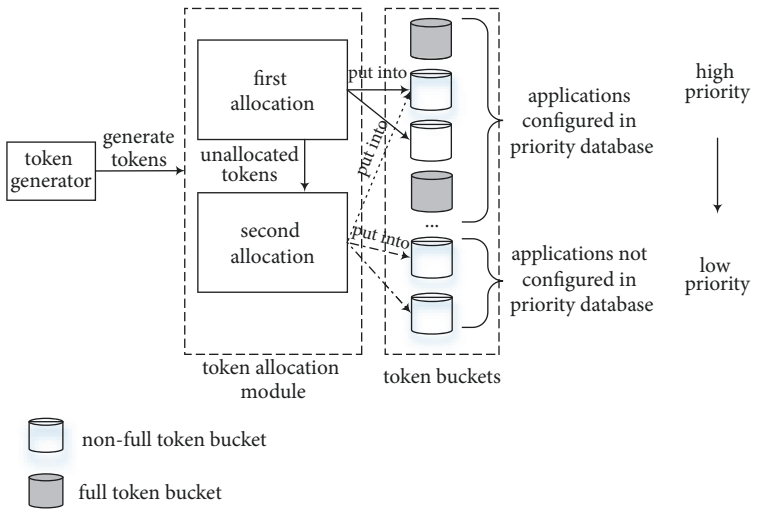

Figure 2: Flowchart of bandwidth dynamic scheduling method.

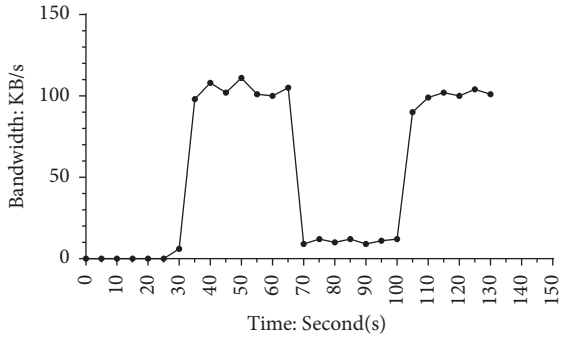

Figure 3: Upload bandwidth control.

QoS for applications. The other method is to preallocate bandwidth for applications. Linksys P-WRT1900ACS router [21] uses this method, which preallocates bandwidth for each port or IP to ensure the bandwidth will not be overconsumed by some applications. The second method can guarantee the QoS for applications, but it has the problem of idle bandwidth and cannot maximize the bandwidth usage.

Tokens are privately owned by applications in traditional token bucket algorithm, which are not shared with other applications. This mechanism can guarantee the QoS for applications but has the problem of idle bandwidth. An improved token bucket algorithm is proposed in this research, which designs a borrowing mechanism to make tokens sharable among applications. A new bandwidth dynamic scheduling method based on the improved token bucket algorithm is proposed, which can avoid the problem of idle bandwidth while guaranteeing the QoS for applications. Figure 3 is the flowchart of the new method, which consists of three components: token generator, token allocation module, and token buckets. Token generator generates tokens and puts them into token buckets through token allocation module. Each application has a privately owned token bucket and all applications share the same token generator and token allocation module. The creation and termination of a process can be monitored by hooking process management functions (such as CreateProcess [22] and TerminateProcess [23]). When a process is started or terminated, its corresponding token bucket is built or destroyed simultaneously. A packet can be transmitted only if its corresponding token bucket has enough tokens. Otherwise, the packet will be buffered and suspended to transmit until there are enough tokens.

To achieve bandwidth dynamic scheduling through the above mechanism, the following three questions should be answered: (1) What packet size each token corresponds to? (2) What is the token generation rate? (3) What is the capacity of each token bucket? For question (1), each 1500-byte packet is associated with a token allowing that the Ethernet MTU (Maximum Transmission Unit) is such size [24]. For question (2), the token generation rate should match the available bandwidth of the current computer. The available bandwidth can be calculated by monitoring the peak transmission speed within a period of time. Allowing the non-real-time characteristic of popular operating systems, the token generation rate is set to 1.2 times the actual bandwidth to avoid the possible token generation delays. For question (3), if an application is listed in the priority database, its token bucket capacity is set to 1.2 times the configured bandwidth to match the total token generation rate. If it is not listed in the priority database, the capacity is set to a preconfigured value.

Token allocation module in Figure 2 is the pivot of bandwidth dynamic scheduling method, which consists of two allocations. In the first allocation, the allocation module will check the token bucket of each running process listed in priority database in descending order of priority. If the bucket is not full, put tokens into it according to the configured bandwidth in the priority database. Otherwise, the tokens will be reserved to the second allocation. In the second allocation, a token borrowing mechanism is designed to realize token sharing.

Input: The tokens generated by generator (L_{token})
Output: The first token allocation policy ($L_{allocation}$)
(1) **FirstAllocation** (L_{token}) {
(2) Obtain the running Web process list $L_{process}$
(3) Query the priorities of processes in $L_{process}$ in the
(4) priority database
(5) Sort $L_{process}$ in descending order of priority
(6) Initialize the first allocation policy $L_{allocation}$
(7) FOR i=0 to COUNT($L_{process}$)-1 {
(8) Mark the bucket of i_{th} process in $L_{process}$ as B_i
(9) IF ($f_{\_currentSize}(B_i) < f_{\_maxSize}(B_i)$) {
(10) // *Non-full bucket*
(11) Get the bandwidth V_i for the corresponding
(12) process in the priority database
(13) IF (V_i != null) { // *Process exists in the priority database*
(14) // T_i *is the remaining capacity of* B_i
(15) $T_i = f_{\_maxSize}(B_i) - f_{\_currentSize}(B_i)$
(16) Initialize the allocation policy (*allocation*) for B_i
(17) // *Set the bucket member of allocation (token ID)*
(18) *allocation*.BUCKET = B_i
(19) // *Set the TOKEN member of allocation (token count)*
(20) // *Add the smaller one between* $1.2 * V_i$ *and* T_i *to the*
(21) // *current token bucket*
(22) *allocation*.TOKEN = *allocation*.TOKEN+MIN(T_i, $1.2 * V_i$)
(23) Add *allocation* into the first allocation policy $L_{allocation}$
(24) Delete the allocated tokens from L_{token}
(25) }
(26) ELSE { // *All processes in priority database checked*
(27) // A *null value of* V_i *means that the corresponding process*
(28) // *of* B_i *does not exist in the priority database and the*
(29) // *remaining processes do not exist in the database*
(30) // *either.These processes won't be allocated any tokens*
(31) // *in the first allocation.*
(32) RETURN $L_{allocation}$
(33) }
(34) }
(35) }
(36) RETURN $L_{allocation}$ // *All processes checked*
(37) }

Algorithm 1: Bandwidth dynamic scheduling method (the first allocation).

The *maxBorrow* parameter is introduced to control how much bandwidth an application can borrow from other applications, which can be set to a larger value when wanting to borrow more bandwidth or set to zero to disable borrowing. *MaxBorrow* will gradually decrease with the transmissions of packets. When the *maxBorrow* for a process decreases to zero, the *maxBorrow* for all the processes with higher or equal priority will be reset to their initial values, while the *maxBorrow* for the other processes will remain unchanged. This means that, when detecting a zero value of *maxBorrow* for some process, the token borrowing privilege will always be granted to all the processes with higher or equal priority simultaneously, and the processes with lower priority will not get such privilege.

The first allocation guarantees that every running process listed in the priority database can obtain the configured bandwidth. The following data structure is introduced to specify the allocation process.

```
struct allocation{
    BUCKET          // ID of token bucket
    TOKEN           // The token count
}
```

Algorithm 1 is the pseudo code for the first allocation. Lines starting with // are comments and all the other lines are valid code. The number at the beginning of each line is the line number. The input of the algorithm is the generated tokens (L_{token}) and the output is the first token allocation policy ($L_{allocation}$).

In Algorithm 1, firstly obtain the running process list ($L_{process}$, *line (2)*), then query the priority database to retrieve the priorities of processes in $L_{process}$ (*lines (3)-(4)*) and sort these processes in descending order of priority (*line (5)*). Initialize the allocation policy ($L_{allocation}$, *line (6)*) and loop through $L_{process}$ (*lines (7)-(35)*). Use functions $f_{\_currentSize}$ and $f_{\_maxSize}$ to get the current and maximal capacity of a token bucket (B_i), respectively, and check whether the bucket is full. If it is not full (*lines (9)-(10)*), query the configured bandwidth (V_i) in the priority database (*lines (11)-(12)*). If V_i is not null, get the remaining capacity of B_i and mark it as T_i (*lines (13)-(15)*). Initialize the allocation policy for B_i (*line (16)*), assign B_i to its BUCKET member and add MIN(T_i, $1.2 * V_i$) to its TOKEN member (*lines (17)-(22)*). After that, add it to the first allocation policy ($L_{allocation}$) and delete the allocated tokens from L_{token} (*lines (23)-(24)*). A null value of V_i represents that its corresponding process is not listed in the priority database. Since $L_{process}$ is looped through in descending order of priority, when finding a process not listed in the priority database, all the remaining processes are not listed in this database either. In this case, stop looping and return $L_{allocation}$ directly (*lines (26)-(33)*). If all the processes in $L_{process}$ have been checked, return $L_{allocation}$ (*line (36)*).

Through the first allocation, all the running processes listed in the priority database have attained tokens and token-based packet transmission privileges. Combining the priority mechanism, the QoS for these processes can be guaranteed.

The unallocated tokens (marked as L'_{token}) in the first allocation will be further allocated to other processes in the second allocation. The token sharing mechanism is introduced to maximize the bandwidth usage during the second allocation. Algorithm 2 is the pseudo code, whose input is the unallocated tokens (L'_{token}) and the output is the second token allocation policy ($L'_{allocation}$).

In Algorithm 2, firstly obtain all the running Web processes not listed in the priority database ($L'_{process}$, *lines (2)-(3)*). These processes do not get any tokens in the first allocation, so that tokens will be allocated to them firstly. Loop through $L'_{process}$ to retrieve the token bucket for each process (B_i, *lines (5)-(6)*). When a nonfull token bucket is found (*line (7)*), get its remaining capacity (T_i, *lines (8)-(9)*). Initialize *allocation* for B_i and set its members (*lines (10)-(15)*), then add it into the second allocation policy (*line (16)*) and remove the allocated tokens from L'_{token} (*line (17)*). Since all the processes in $L'_{process}$ have the same and lowest priority, evenly allocate tokens for them (*lines (20)-(33)*). Firstly, calculate the average count of allocated tokens (*lines (20)-(24)*), then update the token count for the second allocation policy (*lines (25)-(33)*). If all the tokens have been allocated (*line (34)*), return $L'_{allocation}$ directly (*line (35)*). Otherwise, allocate tokens to all the running Web processes (L_p, *lines (37)-(65)*). Loop through L_p to retrieve each token bucket (M_j, *lines (42)-(43)*). If the bucket is not full and its *maxBorrow* ($M_{j\_maxBorrow}$) is greater than zero, allocate tokens for it and add the smallest value among the unallocated token count (COUNT(L'_{token})), the remaining capacity of this bucket (T_j) and its token borrowing count ($M_{j\_maxBorrow}$) to its current bucket count (*lines (44)-(54)*). Add it into the second allocation policy (*line (55)*), and decrease the token borrowing count (*lines (56)-(57)*) and the unallocated token count (*line (58)*), respectively. If all the tokens have been allocated (*line (59)*), return $L'_{allocation}$ (*line (60)*). L'_{token} will be discarded and a null allocation policy will be returned if no bucket satisfies all the above conditions (*line (66)*).

It can be seen that from Algorithm 2, processes not listed in the priority database will be firstly allocated tokens using an average allocation policy, which guarantees that they have equal rights to use bandwidth. Since these processes have the lowest priority, their token bucket capacity is set to a preconfigured value. It is possible that there are still unallocated tokens after this average allocation. These tokens will be further allocated among all the running Web processes regardless of their existence in the priority database. This can maximize the bandwidth usage and avoid the problem of idle bandwidth.

It should be noted that other tool functions and facilities are also needed except those listed in Algorithms 1 and 2. For example, a token generating function should be used to generate tokens periodically; a periodic timer should be used to reset the *maxBorrow* value for each process; a queue should be constructed to buffer the packets in low priorities that cannot be transmitted immediately. These functions and facilities are omitted for the sake of brevity.

4. Experimental Results

Three experiments (Experimental PC settings: Intel i7-3770 CPU, 16G RAM, Windows 7 Professional) were designed to verify the effectiveness of the new method. Experiment 1 verified the capability of bandwidth control. Experiment 2 verified the effectiveness of bandwidth scheduling. Experiment 3 verified the improvement of QoS.

4.1. Capability of Bandwidth Control. *Baidu Netdisk* (a cloud storage application) [25] was used to upload a 1G byte file and its bandwidth usage was illustrated in Figure 3. It can be seen that the upload rate was 100 KB/s at the beginning and it decreased to 10 KB/s when a bandwidth limit of 10 KB/s was applied at the tick of 70 second. It recovered to 100 KB/s gradually after the limit was removed.

Similar results were gained in the file download experiment (Figure 4). These results indicate that the new method has a good capability in bandwidth control.

4.2. Effectiveness Of Bandwidth Dynamic Scheduling. Three applications were used in this experiment: a video conference application (*Fsmeeting*) [26], an online music player (*QQ Music*) [27], and a download manager (*Thunder*) [28]. The priority is *Fsmeeting* > *QQ Music* > *Thunder*. The total download bandwidth for them was set to 200 KB/s and the separate download bandwidth for them was set to 130 KB/s, 50 KB/s, and 20 KB/s, respectively (Table 1). All the three applications were allowed to borrow idle bandwidth. Figure 5 illustrates the experimental results. It can be seen that the three processes consumed the preconfigured bandwidth during the period of 0~50 seconds, which indicates that their

Input: The unallocated tokens after the first allocation (L'_{token})
Output: The second token allocation policy ($L'_{allocation}$)
(1) **SecondAllocation** (L'_{token}) {
(2) Obtain the running Web processes not listed in
(3) the priority database and mark them as $L'_{process}$
(4) Initialize the second allocation policy $L'_{allocation}$
(5) FOR i=0 to COUNT($L'_{process}$)-1 { // *Retrieve each process*
(6) Mark the bucket of i_{th} process in $L'_{process}$ as B_i
(7) IF ($f_{\_currentSize}(B_i) < f_{\_maxSize}(B_i)$) { // *Bucket is not full*
(8) // T_i *is the remaining capacity of* B_i
(9) $T_i = f_{\_maxSize}(B_i) - f_{\_currentSize}(B_i)$
(10) Initialize the allocation policy (*allocation*) for B_i
(11) // *Set the BUCKET member of allocation (token ID)*
(12) *allocation*.BUCKET = B_i
(13) // *Set the TOKEN member of allocation (token count).*
(14) // *This value may be modified in the following steps.*
(15) *allocation*.TOKEN = *allocation*.TOKEN + T_i
(16) Add *allocation* into the second allocation policy $L'_{allocation}$
(17) Delete allocated tokens from L'_{token}
(18) }
(19) }
(20) // *Calculate how many tokens each process can get*
(21) // *using average allocation policy*
(22) IF (COUNT($L'_{allocation}$) > 0) {
(23) AVE = COUNT(L'_{token}) / COUNT($L'_{allocation}$)
(24) }
(25) // *Loop through* $L'_{allocation}$ *to update the token count for*
(26) // *each allocation*
(27) FOR k=0 to COUNT($L'_{allocation}$)-1 {
(28) Mark the k_{th} item of $L'_{allocation}$ as A_k
(29) // *update the token count using the average value*
(30) A_k.TOKEN = A_k.TOKEN +
(31) MIN($f_{\_maxSize}(A_k) - f_{\_currentSize}(A_k)$, AVE)
(32) Delete the allocated tokens from L'_{token}
(33) }
(34) IF (COUNT(L'_{token}) == 0) { // *Token allocation finished*
(35) RETURN $L'_{allocation}$
(36) }
(37) ELSE { // *Still have unallocated tokens*
(38) // *Try to allocate the remaining tokens to all the running*
(39) // *Web processes regardless of their existence in the*
(40) // *priority database through token borrowing*
(41) Get all the running Web process L_p and sort it in descending order of priority
(42) FOR j=0 to COUNT(L_p)-1 {
(43) Mark the bucket of the j_{th} process in L_p as M_j
(44) IF ($f_{\_currentSize}(M_j) < f_{\_maxSize}(M_j)$) { // *Bucket is not full*
(45) Get the *maxBorrow* of the current bucket ($M_{j\_maxBorrow}$)
(46) IF ($M_{j\_maxBorrow}$ > 0) { // *Can borrow more tokens*
(47) // T_j *is the remaining capacity of* M_j
(48) $T_j = f_{\_maxSize}(M_j) - f_{\_currentSize}(M_j)$
(49) Initialize *allocation* for M_j
(50) // *Set the BUCKET member of allocation (token ID)*
(51) *allocation*.BUCKET = M_j
(52) // *Set the TOKEN member of allocation (token count)*
(53) *allocation*.TOKEN = *allocation*.TOKEN +
(54) MIN(COUNT(L'_{token}), T_j, $M_{j\_maxBorrow}$)
(55) Add *allocation* into the second allocation policy
(56) Subtract the count of allocated tokens from the token
(57) borrowing parameter ($M_{j\_maxBorrow}$)
(58) Delete the allocated tokens *from* L'_{token}
(59) IF (COUNT(L'_{token}) == 0) { // *Token allocation finished*

Algorithm 2: Continued.

```
(60)    RETURN L'allocation
(61)    }
(62)    }
(63)    }
(64)    }
(65)  }
(66)  RETURN null
(67) }
```

Algorithm 2: Bandwidth dynamic scheduling method (the second allocation).

Table 1: Bandwidth and priorities configurations for the three applications.

Application	Bandwidth	Priority
Fsmeeting	130KB/s	High
QQ Music	50KB/s	Medium
Thunder	20KB/s	Low

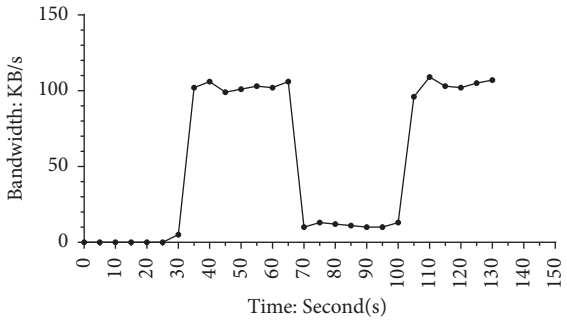

Figure 4: Download bandwidth control.

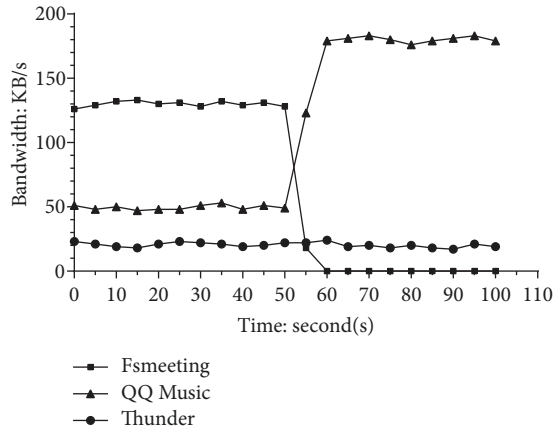

Figure 5: Bandwidth usage for three applications.

bandwidth can be scheduled correctly. After the tick of 50 seconds, *Fsmeeting* was killed and its bandwidth usage decreased to 0 KB/s. The bandwidth usage of *QQ Music* gradually increased to 180 KB/s, while the bandwidth usage of *Thunder* is kept unchanged during this period. These experimental results can verify that the new method has good effectiveness in bandwidth sharing and dynamic scheduling, which can maximize the bandwidth usage without the problem of idle bandwidth.

4.3. Verification of QoS Improvement. A VOIP application (*MicroSIP* [29]) was used to verify the improvement of QoS. The reason for choosing a VOIP application is that such applications are very sensitive to bandwidth and their QoS can be measured through a relatively easy method named Mean Opinion Score (MOS) [30]. Strictly speaking, the QoS for VOIP applications can be influenced by many factors, for example, bandwidth, network delay, packet loss, etc. And bandwidth is not the only influencing factor. In this experiment, the other factors were assumed unchanged and bandwidth was assumed to be the only influencing factor.

Currently there is no relevant research on the impact of local bandwidth scheduling for QoS guarantee in the local computer. Therefore, in this research there was no comparison experiment with the existing methods. Instead, two experiments were designed to verify the improvement of QoS when using the new method or not. Three applications *Baidu Netdisk*, *Thunder,* and *MicroSIP* were used in the two experiments. Table 2 lists their configurations in bandwidth and priorities.

The steps of the two experiments are as follows: download a 1G byte file using *Baidu Netdisk* and *Thunder* simultaneously, and during this period use *MicroSIP* to make calls. The first experiment used the new QoS method and the second one did not use it. Figures 6 and 7 illustrate the MOS scores and the bandwidth usage in two cases.

A piece of music instead of human voice was used when making calls to avoid man-made influence. The call quality was scored through MOS which ranged from 1 to 5. The higher the call quality, the higher the MOS score. Thirty participants were invited to scored their MOS. Figure 6 illustrates the MOS in two experiments. The average MOS is 3.91 and 3.18 when using the new QoS method or not, respectively. The former has a 23% higher score than the latter, which concludes that the QoS can be improved significantly when using the bandwidth dynamic scheduling method.

Figure 7 illustrates the bandwidth usage of *MicroSIP* in the two experiments. It can be seen that the bandwidth usage cannot be guaranteed when not using the QoS method. In this case, three applications competed bandwidth freely and *MicroSIP* did not have any bandwidth guarantee. When using the QoS method, the bandwidth for *MicroSIP* was

Table 2: Bandwidth and priority configurations in two cases.

Application	Using method		Not using method	
	Bandwidth	Priority	Bandwidth	Priority
MicroSIP	100 KB/s	High	null	null
Baidu Netdisk	80 KB/s	Medium	null	null
Thunder	null	Low	null	null

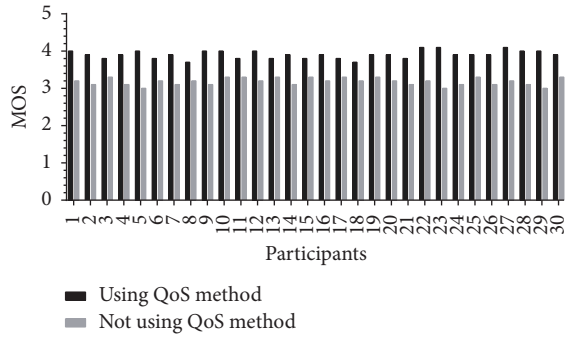

Figure 6: MOS scores of the two experiments.

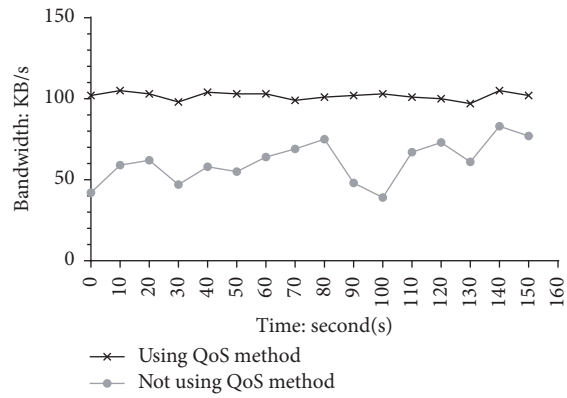

Figure 7: Bandwidth usage for MicroSIP in two cases.

generally stable at 100 KB/s, although the other two applications were performing significant bandwidth-consuming operations. This concludes that, from another perspective, the local bandwidth scheduling method can guarantee the bandwidth for Web applications and can improve their QoS effectively.

5. Conclusion

A new application-level QoS control method based on local bandwidth scheduling was proposed and experimental results verified its effectiveness. The new method has three advantages. (1) It can schedule bandwidth without tedious manual configurations. The configurations for commonly used scenarios can be built in advance and most users can reuse them and make their own extensions. This will simplify the configuration and reduce lots of workload. (2) It schedules bandwidth at application level and can control the bandwidth for applications using dynamic ports. (3) It can be easily integrated into current Web applications without modifying the underlying network.

It should be noted that the existing QoS guarantee methods which work at router or network layer are effective and practically verified. Our method is not a competitive or replaceable relationship with these methods. To the opposite, it is an organic supplement to them and can improve QoS further at application level. The new method only solved the problem of bandwidth scheduling in local computer, which cannot avoid the excessive bandwidth consumption caused by other computers in the same local area network. Further research can be made to solve the problem in this scenario.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (61772078), Key R&D Program of Beijing (D171100001817003), and the Fundamental Research Funds for the Central Universities (YX2014-17).

References

[1] A. Sinaeepourfard and H. Mohamed Hussain, "Comparison of VOIP and PSTN services by statistical analysis," in *Proceedings of the IEEE Student Conference on Research and Development*, pp. 459–461, Piscataway, NJ, USA, December 2012.

[2] "Resource ReSerVation Protocol (RSVP), IETF website [EB/OL]," 2017, https://tools.ietf.org/html/rfc2205.

[3] C. Metz, "Differentiated Services," *IEEE MultiMedia*, vol. 7, no. 3, pp. 84–90, 2000.

[4] P. Papadimitriou and V. Tsaoussidis, "On transport layer mechanisms for real-time QoS," *Journal of Mobile Multimedia Rinton Press*, vol. 1, no. 4, pp. 342–363, 206.

[5] A. Neto, E. Cerqueira, A. Rissato, E. Monteiro, and P. Mendes, "A resource reservation protocol supporting QoS-aware multicast trees for next generation networks," in *Proceedings of the IEEE Symposium on Computers and Communications*, pp. 707–714, Santiago, Portugal, July 2007.

[6] Q. Fu, "A traffic-aware multipath QoS routing strategy," *Jisuanji Xuebao/Chinese Journal of Computers*, vol. 37, no. 10, pp. 2153–2164, 2014.

[7] A. Ayyasamy and K. Venkatachalapathy, "Context aware adaptive fuzzy based QoS routing scheme for streaming services over MANETs," *Wireless Networks*, vol. 21, no. 2, pp. 421–430, 2014.

[8] P. Sondi, D. Gantsou, and S. Lecomte, "Design guidelines for quality of service support in Optimized Link State Routing-based mobile ad hoc networks," *Ad Hoc Networks*, vol. 11, no. 1, pp. 298–323, 2013.

[9] W. Lingrui, *Research about Network Traffic Control on Linux*, University of Science and Technology of China, Beijing, China, 2014.

[10] http://www.utorrent.com, 2017.

[11] https://www.ietf.org/rfc/rfc2474.txt, 2017.

[12] Y. Zhang, S. Liu, R. Zhang et al., "A new multi-service token bucket-shaping scheme based on 802.11e," in *Proceedings of the International Conference on Identification, Information, and Knowledge in the Internet of Things*, pp. 691–696, Zhejiang, China, October 2015.

[13] N. Lin and N. Suo, "Hierarchical and dynamic traffic shaping algorithm based on token buckets," *Modern Computer*, vol. 6, no. 1, pp. 7–9, 2011.

[14] J. Kidambi, D. Ghosal, and B. Mukheqee, "Dynamic token bucket (DTB): A fair bandwidth allocation algorithm for high-speed networks," *Journal of High Speed Networks*, vol. 9, no. 2, pp. 67–87, 2000.

[15] T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, and G. Lipari, "On the integration of application level and resource level QoS control for real-time applications," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 479–491, 2010.

[16] D. G. Balan and D. A. Potorac, "Linux HTB queuing discipline implementations," in *Proceedings of the IEEE International Conference on Networked Digital Technologies*, pp. 122–126, Australia, 2009.

[17] N. Aoki, "A technique of lossless steganography for G. 711," *IEICE Transactions on Communications*, vol. E90-B, no. 11, pp. 3271–3273, 2007.

[18] R. Mohanty, H. Behera S, and K. Patwari, "Priority based dynamic round robin algorithm with intelligent time slice for soft real time systems," *International Journal of Advanced Computer Science & Applications*, vol. 16, no. 1, pp. 54–60, 2011.

[19] *Hook*, Wikipedia website, 2017, https://en.wikipedia.org/wiki/Hooking.

[20] *Windows Filtering Platform*, 2017, https://msdn.microsoft.com/en-us/library/windows/desktop/aa366510(v=vs.85).aspx/.

[21] *Linksys P-WRT1900ACS*, 2017, https://www.linksys.com/cn/p/P-WRT1900ACS/.

[22] *CreateProcess Function*, 2017, https://msdn.microsoft.com/en-us/library/windows/desktop/aa366510(v=vs.85).aspx.

[23] *TerminateProcess Function*, 2017, https://msdn.microsoft.com/en-us/library/windows/desktop/ms686714(v=vs.85).aspx.

[24] *Maximum Transmission Unit*, 2017, https://en.wikipedia.org/wiki/Maximum_transmission_unit.

[25] *Baidu Netdisk*, 2017, https://pan.baidu.com.

[26] *Fsmeeting*, 2017, http://www.fsmeeting.com.

[27] *QQ Music*, 2017, https://y.qq.com.

[28] *Thunder*, 2017, http://dl.xunlei.com.

[29] *MicroSIP*, 2017, http://www.microsip.org.

[30] M. Viswanathan and M. Viswanathanb, "Measuring speech quality for text-to-speech systems: development and assessment of a modified mean opinion score (MOS) scale," *Computer Speech and Language*, vol. 19, no. 1, pp. 55–83, 2005.