Flow Management for SIP Application Servers

Jing Sun
Department of Computer Science and Technology
Tsinghua University
Beijing, China
sunjing05@mails.tsinghua.edu.cn

Jinfeng Hu, Ruixiong Tian, Bo Yang
IBM China Research Lab
Beijing, China
{hujinf, tianruix, boyang}@cn.ibm.com

Abstract—In this paper, we study how to build a front-end flow management system for SIP application servers. This is challenging because of some special characteristics of SIP and SIP applications. 1) SIP flows are well organized into sessions. The session structure should be respected when managing SIP flows. 2) SIP has been adopted by telecom industry, whose applications have more critical QoS requirements than WEB ones. 3) SIP message retransmissions exacerbate the overload situation in case of load bursts; moreover, they may trigger persistent retransmission phenomenon, which retains large response times even after the original burst disappears. To address the combination of these challenges, we propose a novel front-end SIP flow management system FEFM. FEFM integrates concurrency limiting, message scheduling and admission control to achieve overload protection and performance management. It also devises some techniques such as response time prediction, twin-queue scheduling, and retransmission removal to accomplish SLA-oriented improvement, reduce the call rejection rate and banish the persistent retransmission phenomenon. Intensive experiments show that FEFM achieves overload protection in burst period, improves performance significantly, and has the ability to compromise different tradeoffs between throughput and SLA satisfaction.

Index Terms—SIP, Flow Management, Session, SLA

1. INTRODUCTION

Session Initiation Protocol (SIP) [1] has been widely accepted as a major signaling protocol to establish and manage sessions in IP networks. Telecom providers also adopt it in their next generation networks, such as IP Multimedia Subsystem (IMS) of the 3rd Generation Partnership Project (3GPP) in the emerging Universal Mobile Telecommunications System (UMTS) networks [2]. Under such a situation, leading J2EE middleware providers also embrace SIP and integrate corresponding supports (e.g., JSR 116 [3]) into their major products, such as IBM WebSphere and BEA WebLogic.

Building healthy, long-running SIP application servers is not easy. Burst or overload may hurt the performance significantly. Therefore, methods are needed to manage the performance of SIP servers. Though a lot of works have been done toward managing performance of WEB servers, which is based on HTTP protocol, SIP introduces some new challenges.

First, SIP messages are well organized into sessions. Fig. 1 shows a simple SIP session example in which an INVITE transaction creates the session and the BYE transaction terminates it [1]. When a server is overloaded, messages will experience longer response times than usual or even may time out. Timeout of a message that belongs to an existing session might break down the session, which would be a bad experience to end users. Therefore, session integrity should be respected in SIP performance management.

Second, SIP has been adopted by telecom industry in 3G networks and NGN, whose applications demand more critical QoS than WEB ones. For example, in a simple telephony call, the response time of the session-establishing request, i.e., the first (non-retransmitted) INVITE, on the server end, should not exceed a few hundred milliseconds with a very high probability. Here “response time” is defined as the time interval from receiving a session-establishing request to sending out the first synchronous reply. The first synchronous reply denotes the first non-100 response when the server is a proxy or a B2BUA (Back-to-Back User Agent) [1]. This demands that the performance management should consider the SLA-specified response time requirement explicitly and keep as many requests as possible to meet that requirement, especially during the high load period.

Third, SIP messages are usually transmitted by UDP. To improve the transmission reliability, message retransmissions are thereby employed in the protocol design. However, such retransmissions will exacerbate the situation when load burst occurs. Moreover, they might trigger persistent retransmission phenomenon, in which response times could not turn down even after the burst is over.

Figure 1. A simple SIP session.
To address these issues, we designed a front-end flow management system (FEFM) to do overload protection and performance management for back-end SIP servers. What FEFM achieves includes:

1. It queues the session-establishing requests and limits the number of concurrent messages on the servers, which prevents the servers from overload.
2. It predicts session-establishing requests’ response times and priorities those requests that could meet the SLA requirement when scheduling messages.
3. It rejects some session-establishing requests when the load is too high. This protects the performance of those accepted sessions.
4. It removes retransmissions, which lightens the server and banishes the persistent retransmission phenomenon.
5. It allows various tradeoffs between throughput and SLA satisfaction by key parameter adjustment.

We organize the rest of this paper in the following way. First, a profile of SIP server running without flow management is shown in section II, which illustrates the situation that we are facing. Then, section III proposes the architecture and algorithms of FEFM. Section IV shows experiment results. After discussing related works in section V, we close the paper with a brief conclusion in section VI.

II. Real Running Profile

To understand the SIP server performance, we built a SIP application and conducted some performance testing.

The SIP flow is illustrated in figure 1. The SIP application server acts as a UAS. After receiving an INVITE request, it looks up a database to obtain user information of the caller, responds with 180, inserts an “ongoing call” record into the database, and then responds with 200. After receiving an ACK request, it updates the status of the call record in database. After receiving a BYE request, it moves the call record to another table that archives all the finished calls and responds with 200.

The server application is built in WebSphere that has a JSR-116-compliant SIP container. The node running the server has an Intel 2.4G Hz hyper-threaded P4 CPU and 4GB RAM. The operating system is RedHat ES 3 update 3. The database runs on the same node with the server, which is an instance of mysql 5.0. A SIPP (version 1.1 [4]) runs UACs on another node that has the same hardware and operating system with the server node. The two nodes are connected via 1GB Ethernet.

A. Standard running

For Java applications, garbage collection (GC) policy has significant impact on the performance. Therefore, we first conducted some basic experiments to compare the GC policies. Figure 2, 3 and 4 compare the response time results of two experiments with different GC policies, optthruput (the default one) and gencon (the “generational + concurrent” one). Table I shows more statistics. In both experiments, SIPP initializes 200 calls per second (cps) and the server’s JVM maximum

![Figure 2. Response times when using optthruput.](image1)

![Figure 3. Response times when using gencon.](image2)

![Figure 4. CDF comparison of two GC policies.](image3)

![Figure 5. Response times in case of a load burst.](image4)

![Figure 6. CPU utilization in case of a load burst.](image5)

<table>
<thead>
<tr>
<th>Statistic</th>
<th>optthruput</th>
<th>gencon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total calls</td>
<td>360000</td>
<td>360000</td>
</tr>
<tr>
<td>Mean (ms)</td>
<td>24.13</td>
<td>24.98</td>
</tr>
<tr>
<td>99% (ms)</td>
<td>274</td>
<td>169</td>
</tr>
<tr>
<td>&lt; 200s calls</td>
<td>355843 (98.84%)</td>
<td>357724 (99.37%)</td>
</tr>
<tr>
<td>Retransmissions</td>
<td>633</td>
<td>0</td>
</tr>
<tr>
<td>Timeout</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2. Response times when using optthruput.
Figure 3. Response times when using gencon.
Figure 4. CDF comparison of two GC policies.
Figure 5. Response times in case of a load burst.
Figure 6. CPU utilization in case of a load burst.

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the ICC 2007 proceedings.
heap size is 2GB. Figure 2 and 3 show the response time of each call. Figure 4 shows the CDF comparison.

We can see that optthruput has response time spikes every time a GC is launched. Gencon does not have. However, gencon’s average response time is a little larger. If SLA requires a response time smaller than 200ms, then gencon satisfies it better than optthruput. We recommend gencon for telecom applications because they demand a small maximum response time, which can be interpreted as a 99.9%- or 99.99%-percentile value in practice, where gencon performs much better. We also conducted experiments using another GC policy optavgpause, which has a similar result with optthruput.

B. In case of load burst

Load burst is inevitable in reality. When a SIP server encounters a burst, the performance will be hurt. Figure 5 shows the response time result when the SIP server experiences a burst. Figure 6 shows the corresponding CPU utilization. In the experiment, the offered load is originally 200cps and surges to 400cps after 240s (implemented by starting SIPP with 200cps on another node). The burst lasts 120s, after which the load drops back to 200cps and continues for 540s.

During the first 240s, the server runs healthily, which responds the requests timely. Since the burst starts, the response times increase continuously because the requests accumulate in the server’s queue. When the burst is over, response times are expected to drop back to the normal, but they do not. This is because in the burst period, messages are retransmitted many times. According to the specification, UAC keeps retransmitting an INVITE request if it does not get a response in T. T commonly starts at 0.5 seconds and doubles each time a retransmission is sent. This procedure stops if no response is got in $T^*$ (32s by default), returning to the transaction user a failure.

From figure 5 we can see that at the time when the burst is just over, the response time is about 3s, which indicates that each request has been retransmitted twice. Then, each request is actually executed by the server three times. Such a load is more than what the server can sustain. Therefore, the request queue will continue building up (more than linearly) until the server crashes. We call such a phenomenon “persistent retransmission”. Specifically, this means that retransmissions persist or even go up after the original factor that triggered the retransmissions disappears. When using TCP as the transmission protocol, persistent retransmission does not happen. However, considering that adopting TCP would sacrifice a large amount of throughput, most SIP applications prefer UDP transmission.

III. FEFM ARCHITECTURE AND ALGORITHMS

In the real cluster deployment, it is very common to use a special node sitting in front of the application servers which acts as a reversed proxy for HTTP and stateless proxy for SIP. It is used for some special functions such as connection maintenance, flow management, attack detection and reaction, load balancing, etc. We propose to design SIP flow management on this node to achieve the following goals.
1. Preventing back-end servers from overload;
2. Making more requests meet the SLA requirement;
3. Rejecting new sessions properly when the offered load is higher than server’s capacity;
4. Allowing tradeoffs between throughput and SLA satisfaction; and
5. Banishing persistent retransmissions.

The system is simply named FEFM (Front-End Flow Management) whose architecture is depicted in figure 7.

When a new request comes, FEFM first classifies it. If it belongs to an ongoing session, it will be directly sent to the server. Otherwise, it will be put into a new-request queue that buffers the session-establishing requests. A throttle is used to limit the total number of the concurrent requests that are being executed on the server to prevent the server from overload. When the server has room for more request(s), the throttle will ask the selector to deliver one or more session-establishing requests into the server. The latter selects them from the header of the new-request queue or the header of the old-request queue. The scheduling algorithm between these two queues will be described in subsection C. The timestamps of each request and its reply when they pass through the throttle will be noted in the recorder. The reply of a non-INVITE
request is defined as its response when the server acts as a UAS or the corresponding outbound request when the server acts as a proxy or a B2BUA. These timestamps help the selector make its scheduling decisions. The parameters used in FEFM are listed in Table II.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Number of current requests running on the server</td>
</tr>
<tr>
<td>N</td>
<td>Upper limit of n, used by the throttle</td>
</tr>
<tr>
<td>D</td>
<td>Response time required by SLA</td>
</tr>
<tr>
<td>D_t</td>
<td>Threshold for the new-request queue</td>
</tr>
<tr>
<td>D_o</td>
<td>Threshold for the old-request queue</td>
</tr>
<tr>
<td>T_e</td>
<td>Last request’s real execution time</td>
</tr>
<tr>
<td>T_w</td>
<td>Waiting time of a request</td>
</tr>
<tr>
<td>( \mu_w )</td>
<td>Predicted execution time of a request</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>Weight of the exponentially moving average</td>
</tr>
</tbody>
</table>

A. Overload protection

To control the load upon the server, FEFM limits the number of the requests being concurrently executed on the server (noted outstanding requests). The throttle sets a threshold \( N \) and observes the number of the outstanding requests at runtime, noted as \( n \). After a request is sent to the server, whether it is a session-establishing request or not, the throttle increases \( n \) by 1. When the reply of a request is received, it decreases \( n \) by 1. For those requests with no response, such as ACKs, a timer is set to be triggered periodically (in our system, it is \( D/2 \), where \( D \) is the response time requirement documented in SLA). It decreases \( n \) by the number of such non-response requests that are sent to the server in the last interval. Regarding a request that should get a response but does not, after a long time (\( 2/D \) in our system), it expires and \( n \) is thereby decreased by 1. Every time \( n \) is decreased, the throttle checks if \( n < N \) holds. If so, it notifies the selector to deliver \( N-n \) new session-establishing requests to the server.

It should be noted here that setting \( N \) does not mean that the number of concurrent requests would be never beyond \( N \) because all the non-INVITE requests are admitted into the server immediately without queuing. However, once \( n \) is over \( N \), no session-establishing request will be admitted into the server any more. Such a situation will persist until some requests get their replies or expire, which renders \( n \) to drop down.

B. Retransmission removal

As shown in section II, the retransmissions that are originally employed to improve the reliability will hurt the performance largely when the server encounters a load burst. Moreover, it may lead to persistent retransmission. In FEFM, we remove unnecessary retransmissions to lighten the server load and banish the persistent retransmission phenomenon.

The retransmission removal is implemented as follows. For a request that has already been delivered to the server but has neither got its reply nor expired, FEFM keeps it in the recorder. When FEFM receives a new request, it first checks if it is already in the recorder, the new-request queue, or the old-request queue. If so, it is deemed as a retransmission and dropped immediately. After a request gets its reply from the server or expires, it is removed from the recorder. After that, its retransmissions will be treated as a non-retransmitted request and not be dropped. Therefore, the retransmission removal mechanism scarifies the transmission reliability very slightly.

C. Response time management

Generally, a SLA imposes response time requirements on many kinds of requests. In FEFM, we focus our management only on session-establishing requests. Other requests are admitted into the server without queuing, so they must have better response times. Such a design is because 1) the response time of session-establishing request is a part of the end-to-end “post-dialing delay” (a.k.a. “post-selection delay”), which is a highly important QoS metric in telecom standards [5][6], and 2) queuing or even rejecting other requests takes the risk of cutting off ongoing sessions, i.e., call failures, which should be intensively avoided in telecom services.

Among all session-establishing requests, FEFM prefers admitting those that could meet the SLA requirement to others. To achieve this, estimation of each session-establishing request’s response time is needed. Note that response time is composed of \( T_w \) (the time that the request waits in the queues in FEFM) and \( \cdot \mu_w \) (its predicted execution time on the server). \( T_w \) is measured by the recorder (according to the time when the request is received by the proxy and the time when it is about to be delivered to the server). \( \cdot \mu_w \) is computed by an exponentially moving average. Every time a session-establishing request gets its reply, \( \cdot \mu_w \) is updated using the following equation.

\[
\cdot \mu_w = a \cdot T_e + (1-a) \cdot \cdot \mu_{w-1}
\]  

\( T_e \) is the request’s execution time measured by the recorder and \( a \) is a weight. Such a moving average is easy to calculate. It does not need to keep the past execution times, but does take them into account. The weight of past execution times is \( a \). The larger \( a \) is, the more responsive \( \cdot \mu_w \) is. Our experiments show that in most cases, an \( a \) around 0.5 works well.

The selector checks the header of the new-request queue when it is asked by the throttle to deliver a new session-establishing request. It predicts the request’s response time by \((T_w + \cdot \mu_w)\). If it is less than \( D_t \), a preset threshold, the request is then delivered to the server; otherwise, the request is moved to the old-request queue and the selector then turns back to check the new header of the new-request queue. Such a process continues until the selector delivers a request from the new-request queue or the new-request queue turns empty. If the
latter happens, the selector then turns to check the header of the old-request queue. If it’s predicted response time is less than $D_2$, another preset threshold, it will be delivered into the server; otherwise, it will be rejected by a “503 Service Unavailable” response. After that, the selector turns back to check the new header of the old-request queue. Such a process continues until a request is delivered from the old-request queue or the old-request queue turns empty. If the latter happens, this round of selection terminates with no request being delivered. When the selector is asked to deliver $N - n$ new session-establishing requests, it runs $N-n$ rounds of the above selection.

As mentioned in section II A, the GC will cause response time spikes. The Gencon GC policy can minimize the effects, but $\sqrt{D}$, the prediction of executing time on server, will be deviated too much, once a global GC occurred. The executing time of the requests running into GC should be excluded from moving average calculation. A recorder continuously monitors the time interval of two consecutive responses received from server. If the value suddenly increases too much over a threshold that is set as 100 ms experimentally, we presume a GC just completed on the server, and the response is the first affected request’s response. Therefore, any request which timestamp is between the timestamp of the request related to the last received response and the timestamp of the current request preparing to send will be affected by GC, so that its executing time will not be used to calculate the moving average. This method prevents amount of requests that could meet the SLA requirement from being thrown into the old-request queue due to the improper (much larger) estimation of $\sqrt{D}$.

D. The twin-queue scheduling

FEFM uses the twin-queue scheduling, rather than a simple FIFO queue, in order to consider the throughput and the SLA satisfaction simultaneously. In practice, requests do not come evenly (it is more reasonable to assume that the incoming rate of session-establishing requests conforms to a Poisson distribution). By the twin-queue scheduling method, those requests that accumulated in the old-request queue during a peak would have a second chance to be delivered into the server during the following off-peak time. Therefore, such a method helps maintain a stable throughput.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>No FEFM</th>
<th>$N=80$</th>
<th>$N=100$</th>
<th>$N=200$</th>
<th>$N=300$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean (ms)</td>
<td>4494.57</td>
<td>95.84</td>
<td>87.63</td>
<td>287.34</td>
<td>320.95</td>
</tr>
<tr>
<td>Max (ms)</td>
<td>18780</td>
<td>2300</td>
<td>2862</td>
<td>6266</td>
<td>5347</td>
</tr>
<tr>
<td>95% (ms)</td>
<td>14185</td>
<td>209</td>
<td>200</td>
<td>2042</td>
<td>2177</td>
</tr>
<tr>
<td>&lt; 200 ms (calls)</td>
<td>50445</td>
<td>172905</td>
<td>179535</td>
<td>158886</td>
<td>156300</td>
</tr>
<tr>
<td>Rejected calls</td>
<td>0</td>
<td>20655</td>
<td>15213</td>
<td>11819</td>
<td>12518</td>
</tr>
<tr>
<td>Calls moved to old req. queue</td>
<td>N/A</td>
<td>26915</td>
<td>20831</td>
<td>42355</td>
<td>47213</td>
</tr>
</tbody>
</table>

$D = 200$ ms, $D_2 = 2000$ ms

<table>
<thead>
<tr>
<th>Statistic</th>
<th>No FEFM</th>
<th>$D_2=500$</th>
<th>$D_2=1000$</th>
<th>$D_2=2000$</th>
<th>$D_2=3000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean (ms)</td>
<td>4494.57</td>
<td>55.53</td>
<td>66.53</td>
<td>87.63</td>
<td>118.94</td>
</tr>
<tr>
<td>Max (ms)</td>
<td>18780</td>
<td>1463</td>
<td>1963</td>
<td>2862</td>
<td>3907</td>
</tr>
<tr>
<td>95% (ms)</td>
<td>14185</td>
<td>201</td>
<td>213</td>
<td>200</td>
<td>207</td>
</tr>
<tr>
<td>&lt; 200 ms (calls)</td>
<td>50445</td>
<td>178501</td>
<td>178439</td>
<td>179535</td>
<td>179365</td>
</tr>
<tr>
<td>Rejected calls</td>
<td>0</td>
<td>16311</td>
<td>15168</td>
<td>15213</td>
<td>14764</td>
</tr>
<tr>
<td>Calls moved to old req. queue</td>
<td>N/A</td>
<td>21736</td>
<td>21580</td>
<td>20831</td>
<td>21485</td>
</tr>
</tbody>
</table>

$D = 200$ ms, $N = 100$
underestimation could happen. Setting $D_1$ as the SLA requirement would move some requests into old-request queue by mistake due to overestimation. Lowering $D_1$ reduces this possibility. Of course it increases the possibility of mistakes from underestimation: moving more requests to the old-request queue which fail to meet the SLA requirement if they were finally delivered to the server. But this is not a big problem because in practice, we often have some room to tolerate it. Most SLAs do not impose a strict requirement upon all the requests. They commonly provide a percentile requirement, e.g., a 95-percentile value. $D_2$ should be larger than the SLA requirement. But we don’t recommend a very large $D_2$ because that indicates long post-dialing delays, which is a bad experience for end users.

IV. EXPERIMENT RESULTS

We implement FEFM in Java, based on a JSR-32-compliant SIP stack [7] that is the same with the one used in the WebSphere SIP container. The node running FEFM has the same hardware and operating system with the server. We ran the burst scenario in section II again, with all the messages traversing FEFM between the SIPP and the server. The parameters are shown in table III.

A. FEFM running results

Figure 8 and 9 show the response time and server CPU utilization of the session-establishing requests in the experiment. Figure 10 is the response time CDF comparison between this experiment and the one in section II.B. More statistics can be found in table IV (column 4). Interesting observation includes:

1. During the burst, response times are almost limited to beneath 2s. The cost is 15213 calls being rejected, accounting for 31.7% of all the calls in the burst period.
2. 179535 calls meet the SLA requirement, accounting for 95.1% of all the admitted calls.
3. During the burst, 20831 session-establishing requests are moved to the old request queue. 5618 of them are finally delivered to the server, most of which have a response time close to $D_2$ (2000ms).
4. After the burst, there is no persistent retransmission. Both the response time and the CPU utilization fall down to the normal.

B. The impact of the key parameters

The most important parameter in FEFM is the concurrency limit number $N$. Different $N$ results in different number of rejected calls, different throughput, and different SLA satisfaction (defined as the fraction of the session-establishing requests that meet the SLA requirement). We vary $N$ and compare the results in table IV. Response times are shown in figure 11.

When $N$ is too small or too large, both the throughput and the SLA satisfaction are low. When in a medium range, $N$ essentially compromises tradeoffs between the throughput and the SLA satisfaction.

Setting $N$ too small (e.g., 80) reduces the server’s service
rate because the server’s resources (CPU, I/O capacity, etc.) cannot be fully utilized. Then, the session-establishing requests have to wait longer in FEFM. This makes more of them be moved to the old request queue and be rejected finally after waiting longer than \(D_2\). Because fewer calls are admitted, fewer would satisfy the SLA requirement.

Setting \(N\) too large (e.g., 300) also reduces the server’s service rate. Note that on the server each request triggers a SIP servlet [3], which is served by an individual thread. Therefore, a large amount of concurrent requests indicate high thread scheduling cost and drastic resource contention, which finally decrease the throughput and the SLA satisfaction.

Comparing the situations where \(N=100\) and \(N=200\), we can see that there are different tradeoffs between throughput and SLA satisfaction. \(N=200\) has larger throughput because resources are utilized more sufficiently. However, as more requests are competing for resources, the execution times turn longer, which decreases the SLA satisfaction.

We also vary \(D_2\) to see its influence on performance. Table V and figure 12 show the result. It is interesting to see that setting larger \(D_2\) almost does not change the number of the requests that are moved to the old request queue, and therefore does not change the SLA satisfaction because almost all the requests delivered from the old request queue cannot meet the SLA requirement. For those requests in the old request queue, a larger \(D_2\) makes more of them delivered to the server finally. But the fraction is rather small and the price is much longer response time. Therefore, we do not recommend large \(D_2\). From our experience, any value in the range from twice the SLA requirement to one second is rather good.

V. RELATED WORKS

To the best of our knowledge, there is not much research work on SIP QoS management. [8] assesses the QoS of SIP-based mobile service, but does not provide methods for improvement. In [9], an admission controller is built based on application specific policy and call authorization status. However, it focuses on the interaction of SIP authorization process and admission controller, rather than how to use admission control to improve SIP performance. [10] uses virtual SIP links (VSL) to build an overlay, upon which QoS is described and guaranteed for SIP applications. This method demands that the application should be developed compliant with the VSL specification.

Our work is more inspired by some techniques in WEB flow management. [11] first uses session-based admission control to achieve overload protection. However, it does not manage the performance. The response time prediction method has also been adopted by some previous works, such as [12]. FEFM integrates these techniques together with some other novel techniques such as retransmission removal, twin-queue scheduling, and concurrency limit to build a well-performing SIP flow management system. There are also some systems managing response time based on runtime feedback [13] [14]. We don’t choose that way because it is not suitable for session-based flows in which a session-establishing request indicates a number of proceeding requests in future.

VI. CONCLUSION

We study the problem of flow management for SIP application servers. The challenge comes from the combination of some SIP characteristics such as session-based flow, critical SLA requirement and persistent retransmission. We propose FEFM to address it, which integrates the techniques of currency limit, session-based admission control, twin-queue scheduling and retransmission removal. Experiments show that such a system can prevent the system from overload, increase the amount of requests that satisfy the SLA, banish the persistent retransmission phenomenon and compromise tradeoffs between throughput and SLA satisfaction. An integrated flow management should also provide two more functionalities than FEFM, load balancing and service differentiation. They also encounter new challenges when facing SIP traffics. Addressing them is our work in the next step.

REFERENCES