AWAIT: Efficient Overload Management for Busy Multi-tier Web Services under Bursty Workloads [⋆]

L. Lu[1], L. Cherkasova[2], V. de Nitto Personé[3], N. Mi[4], and E. Smirni[1]

[1] College of William and Mary, Williamsburg, VA 23187, USA
{llu,esmirni}@cs.wm.edu
[2] Hewlett-Packard Laboratories, Palo Alto, CA 94304, USA
lucy.cherkasova@hp.com
[3] Universitá degli Studi di Roma "Tor Vergata", Rome, Italy
denitto@udiroma.it
[4] Electrical and Computer Engineering, Northeastern University, Boston, MA.
ningfang@ece.northeastern.edu

Abstract. The problem of service differentiation and admission control in web services that utilize a multi-tier architecture is more challenging than in a single-tiered one, especially in the presence of bursty conditions, i.e., when arrivals of user web sessions to the system are characterized by temporal surges in their arrival intensities and demands. We demonstrate that classic techniques for a session based admission control that are triggered by threshold violations are ineffective under bursty workload conditions, as user-perceived performance metrics rapidly and dramatically deteriorate, inadvertently leading the system to reject requests from already accepted user sessions, resulting in business loss. Here, as a solution for service differentiation of accepted user sessions we promote a methodology that is based on blocking, i.e., when the system operates in overload, requests from accepted sessions are not rejected but are instead stored in a blocking queue that effectively acts as a waiting room. The requests in the blocking queue implicitly become of higher priority and are served immediately after load subsides. Residence in the blocking queue comes with a performance cost as blocking time adds to the perceived end-to-end user response time. We present a novel autonomic session based admission control policy, called *AWAIT*, that adaptively adjusts the capacity of the blocking queue as a function of workload burstiness in order to meet predefined user service level objectives while keeping the portion of aborted accepted sessions to a minimum. Detailed simulations illustrate the effectiveness of *AWAIT* under different workload burstiness profiles and therefore strongly argue for its effectiveness.

1 Introduction

One of the most challenging problems for public Internet and e-commerce sites is the delivery of performance targets to users given the unpredictability of Web accesses. As Internet services become indispensable both for businesses and personal productivity, the efficient management of Internet services under periods where the system is overloaded or simply highly variable, is of critical importance. There is a host of solutions to maintain user-perceived performance levels in the form of service-level objectives (SLOs) that focus mainly on admission control and/or techniques for service differentiation that are threshold based [13, 6, 7, 19] but their effectiveness can be compromised if the workload is *bursty*, i.e., it is characterized by sudden temporal "surges" in the intensity of user arrivals and user demands. While capacity planning of systems under bursty workload conditions has been recently demonstrated as critical for business success [22, 23],

⋆ This work was partially supported by the National Science Foundation under grants CNS-0720699, CCF-0811417, and CCF-0937925.

the problem of efficient admission control and service differentiation under temporal workload bursts remains largely unexplored.

To get the intuition why threshold, usage-based techniques may not be effective if the system is subject to bursty conditions, let us consider a system that provides web services and which is built according to the widely used multi-tiered paradigm. Typically, access to a web service occurs in the form of a *session* consisting of many individual requests. For a customer trying to place an order, or a retailer trying to make a sale, the real measure of a web server performance is its ability to process the *entire* sequence of requests needed to complete a transaction. Session-based admission control (SBAC) has been proposed as a solution to the above problem [13] and its gist can be summarized as follows: the system accepts a new session only when the system has enough capacity to process all future requests related to the session, i.e., the system can guarantee that the session completes successfully. If the system is functioning near its capacity, a new session will be rejected (or redirected to another server if one is available).

The original session-based admission control (SBAC) [13] is proposed for a single-tier web server, and its implementation is usage-based. SBAC accepts a new session only when the server CPU utilization is below a certain threshold. However, burstiness in the user arrival flows results in sudden, nearly simultaneous arrivals of requests in the system. The experiments presented in [23] show that under bursty arrivals SBAC is ineffective in maintaining a low ratio of aborted sessions due to a slow reaction to bursts.

Conventional wisdom suggests that the original session-based admission control can be extended for a multi-tiered system in a straightforward way: it should simply be employed at the bottleneck tier. Yet, if burstiness exists in the flows of a multi-tiered system (irrespective of its source, in the arrivals or service) then burstiness triggers the phenomenon of persistent bottleneck switch, i.e., the bottleneck continuously shifts to another tier [22], making control at the bottleneck tier an elusive task.

In this paper, we depart from threshold usage-based policies, and instead we dynamically control the number and the type of user requests (is it a new or an already accepted session?) admitted for processing into the multi-tier system. When the system enters the overload state, we advocate request buffering from the already accepted sessions in a so-called "blocking" queue, that effectively acts as a waiting room. This blocking queue differentiates among the requests of already accepted sessions to those of new sessions, and implicitly gives them higher priority. To this end, we borrow ideas from the theory of queueing networks with blocking [5, 25]. Blocking of accepted sessions during workload surges may be very effective in differentiating accepted sessions from new sessions, but the performance of accepted sessions is still directly bound by the time the requests spent in the blocking queue. That is, if the time spent is so long that results in SLO violations, it is desirable to limit the capacity of the blocking queue in order to bound the user end-to-end time. We perform a sensitivity study to explore the different fixed blocking queue limits under a variety of burstiness profiles and conclude that the effectiveness of blocking is strongly related to the workload burstiness. To address this issue, we propose a parameter-free, autonomic session-based admission control policy called *AWAIT* that adjusts the blocking queue capacity in response to workload burstiness. We perform detailed simulations using the parameterized TPC-W benchmark with extended functionality for generating bursty session arrivals [23] to demonstrate the effectiveness and robustness of the new strategy. *AWAIT* supports a simple and inexpensive implementation. It does not require significant changes or modifications to the existing Internet infrastructure, and at the same time, it significantly improves the performance of overloaded multi-tier web sites.

This paper is organized as follows. Section 2 presents results that motivate this work. Section 3 presents the admission control algorithm and illustrates its robustness under different burstiness profiles by showing that it consistently meets the sought after performance goals while optimizing its performance targets. Section 4 positions this contribution within the context of related work. Section 5 summarizes the paper.

2 Capacity Planning and Admission Control

In this section, we present a short case study that illustrates how burstiness may impact in an unexpected way the performance of admission control. The basic model of an e-commerce site that we use in this paper is based on the TPC-W benchmark that is implemented as a typical multi-tier application which consists of a web server, an application server, and a back-end database. The web server and the application server reside usually within the same physical server, which is called front sever. After a new session connection is generated, client requests circulate among the front and database server before they are sent back to the client. After a request is sent back, the client spends an average think time E[Z] before sending the following request. A session completes after the client has generated a series of requests.

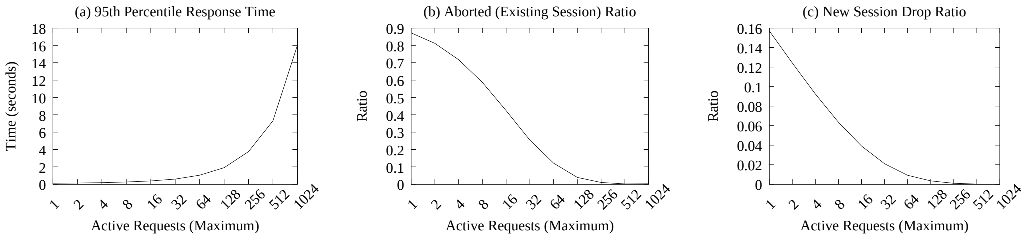

Fig. 1. Capacity Planning study for SBAC under exponential (i.e., not bursty) new session arrivals. Performance measures are presented as a function of the maximum number of active requests in the system.

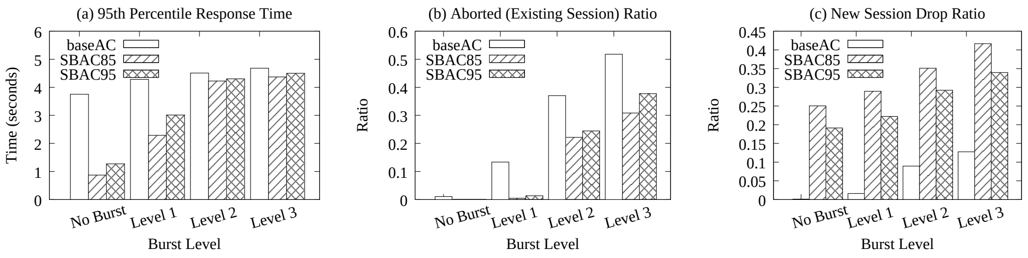

Fig. 2. Three different burstiness profiles. The capacity planning results and SLO targets are now violated. It appears that a queue size of 256 (i.e., maximum active requests for the *baseAC* configuration) is not sufficient to meet SLO requirements.

Overload management is a critical business requirement for today's Internet services. A common approach to handle overload is to apply specific resource limits that typically bound the number of simultaneous socket connections or threads. For example, in traditional web servers that employ thread-per-connection implementation, the server configuration specifies the number of processes (and connections) that are allocated for admitting the user requests. As an example, in the Apache web server [4], when all the server threads are busy, the system stops accepting new connections. The same principle applies for providing the basic overload protection in multi-tier applications. The system administrators may set limits on the number of simultaneous client sessions (we call them *active requests*) in the system. Limiting the active requests is critical for quality of service: setting this limit too low results in achieving a good response time but at a price of lower system throughput (and a high number of dropped user sessions). Setting this limit too high may lead to a better throughput and reduced drop rates at a price of a much higher response time.

Capacity planning is routinely used to determine the base number of active requests in order to strike a balance among the expected customer response times and dropped sessions. Figure 1 illustrates the results of a capacity planning study that models a typical TPC-W 2-tier implementation (i.e., a front server and a database server). The TPC-W defines 14 transactions, each of which can be generally classified as "browsing" or "ordering". Here, we assume that we use the ordering mix, that consists of 50% browsing and 50% ordering transactions. Request service times in the front and database servers are derived using the models presented in [22] that have been shown to capture very accurately the performance and behavior of multi-tier applications. Consistent with the specifications of the TPC-W benchmark, the average user think time is equal to 7 seconds, exponentially distributed. Inter-arrival times of new sessions are assumed to be exponentially distributed, i.e., there is no burstiness in the arrival stream of new sessions.

Each session consists of a sequence of requests (i.e., essentially visit "rounds" to the front and database server that define the a session length) that is uniformly distributed with parameters 5 and 35, that is with expected mean equal to 20.

It is a typical situation when after a certain waiting time an impatient client might "click again" and reissue the original request. Client request timeouts and retries can be added to our model to reflect a more complex and realistic scenario but according to a sensitivity analysis in [13] this will just decrease the useful system throughput (due to the processing overhead of these additional requests) but does not fundamentally change the results of our study. In this paper, we use a simplified model without request timeouts and retries in order to focus on the effects of burstiness.

Figure 1(a) illustrates the 95th percentile of user end-to-end response time as a function of the predefined value of the maximum active requests in the system. Figures 1(b) and 1(c) present the aborted rate of existing sessions and the drop rate of new sessions, respectively, as a function of the allowed active requests. The figure suggests that given this TPC-W mix, one may use 256 as the recommended limit on active requests, since this value strikes a good balance among all desired measures. In all experiments in the remaining of this paper we set the limit on active requests equal to 256.

Session-based admission control (SBAC) [13] is a very effective policy for web servers and is based on monitoring the CPU utilization of the web server. SBAC accepts a new session only when the system utilization is below a certain threshold, to guarantee a successful session completion. If the observed utilization is above a specified threshold, then for the next time interval, the admission controller rejects all new sessions and only serves requests from already admitted sessions. Once the observed utilization drops below the given threshold, the admission controller changes its policy for the next time interval and begins admitting and processing new sessions again. A web server employs a configurable size "listen queue" for buffering the incoming requests. If requests from sessions that are already accepted arrive when the queue is full, then they are aborted. The *useful throughput* of the system is measured as a function of the number of completed sessions. Aborted requests of already accepted sessions are highly undesirable because they compromise the server's ability to process all requests needed to complete a transaction and result in wasted system resources.

We have implemented the SBAC mechanism in a simulation model of a client-server system that is built according to the TPC-W specifications. The SBAC mechanism uses a front server utilization threshold for admitting new sessions. [5] Figure 2 illustrates the ineffectiveness of the threshold-based techniques in presence of bursty arrivals. We compare the results of two different admission control strategies. A first strategy (called *baseAC*) employs a traditional overload control based on admitting a fixed, predefined number of active requests for processing. Here, we set *ActiveRequests = 256* as suggested by capacity planning (see Figure 1). The second strategy is SBAC where the front server utilization threshold is set to 85% and 95% respectively. The three burstiness profiles that we used here are further discussed and described in Section 3.2.

Figure 2(a) illustrates the 95th percentile of user response time. SBAC is effective in maintaining good response times under bursty arrivals but at the expense of a relatively high ratio of aborted sessions as well as a high ratio of rejected new sessions, see Figure2(b) and 2(c). The *baseAC* strategy does not differentiate between the requests from new and existing sessions and this leads to a very high ratio of aborted sessions. While both of these threshold-based strategies might be a reasonable choice under non-bursty traffic, they clearly exhibit their deficiencies under bursty traffic conditions. This simple experiment shows that the admission control mechanism has to take traffic burstiness into account and adapt the system configuration and/or thresholds in order to effectively deal with bursty traffic conditions. In the next section, we present a new algorithm that effectively deals with the above problem.

3 *AWAIT* Algorithm

In this section, we describe *AWAIT*, a novel session-based admission control algorithm that aims to provide an additional support for dealing with bursty session arrivals. *AWAIT* has two different mechanisms to regulate request acceptance for processing. The first mechanism uses a counter of *ActiveRequests* that is defined according to capacity planning for achieving a given SLO for response time. Until this counter reaches its maximum any incoming request is accepted, this request may represent a new session or it may belong to an already accepted session. The second mechanism uses a special queue, called *blocking queue*, which is created to store the requests from already accepted sessions after the number of *ActiveRequests* reaches its maximum capacity. Via this mechanism, the *AWAIT* controller rejects new session requests if *ActiveRequests* is maxed out but the system still admits requests from earlier accepted sessions. When the blocking queue becomes full, then incoming requests from accepted sessions are unavoidably aborted. This is undesirable because it leads to business loss.

The capacity of the blocking queue is a critical parameter for the performance of the accepted sessions since the time spent there contributes to the user end-to-end time, thus may violate the target SLOs. The larger the capacity of the blocking queue, the longer the contribution of the time waiting there to the user end-to-end time. Similarly, the larger the capacity of the blocking queue, the smaller the expected aborted ratio of accepted requests. Striking a good balance between these two conflicting measures is the purpose of *AWAIT*.

[5] For the TPC-W testbed used in our experiments of the ordering mix, SBAC uses the CPU utilization of the front server because the front server is the system bottleneck for this particular mix. In general, admission should be based on the utilization of the bottleneck resource, e.g., if the DB tier is a bottleneck then its CPU utilization should be used for SBAC decisions.

To ease the presentation of *AWAIT*, we first present a static version that considers a fixed blocking queue size. In the adaptive version of *AWAIT*, the size of this blocking queue is autonomically adjusted according to the burstiness of the workload while ensuring that the response time SLOs are met.

3.1 Static *AWAIT*

To formally describe the *AWAIT* algorithm, we introduce the following notions:

- *New session request* – a request that is generated by a new client (i.e., it is a first request in a new session);
- *Accepted session request* – a request that is issued by a client within an already accepted session;
- *ActiveRequests* – a counter that reflects the number of accepted requests which are currently in processing by the system. These active requests could be either of new sessions or of already accepted sessions. The maximum value for this counter is set to a value defined by capacity planning (see Section 2). Let us denote this value as A;
- *BlockedRequests* – a counter that reflects the number of blocked requests which are received from the clients of already accepted sessions and which are stored in the *BlockingQueue*. Note this difference: the blocking queue stores requests from already accepted sessions only. Let B denote the maximum value of this counter that also defines the capacity of this queue;
- *AdmitNew* – a boolean variable that defines whether a new session can be accepted by the system. If $AdmitNew = 1$ then a new session can be accepted by the system. If $AdmitNew = 0$ then all the new sessions are rejected by the system;

Now, we describe the iteration steps of the algorithm. Let a new request *req* arrive for processing. The system can be in one of the following states.

- $AdmitNew = 1$ and $ActiveRequests < A$.
  This state corresponds to normal system processing when there is enough system capacity for processing requests from new sessions as well as requests from already accepted sessions. Therefore, independent on the request type *req* is accepted for processing and the counter *ActiveRequests* increases by one. When this counter reaches its maximum value A, then $AdmitNew = 0$, and this corresponds to a new system state when any requests from new sessions are rejected.
- $AdmitNew = 0$ and $BlockedRequests < B$.
  In this state the incoming requests are treated differently depending on their type. If the incoming request is from a new session then it is rejected. If it is part to an already accepted session, then it is stored in the $BlockingQueue$ and the queue's counter is updated.
- $AdmitNew = 0$ and $BlockedRequests = B$.
  This state reflects to the situation when $BlockedRequests$ has reached its maximum value B. Any incoming request, independent on its type, is rejected. If the request comes from an already accepted session, then its entire session is *aborted.*

Now, we describe how the system counters $ActiveRequests$ and $BlockedRequests$ are updated when a processed request leaves the system, i.e., the reply is sent to the client. The system can be in one of the following states (similar to the states described above).

- If $ActiveRequests < A$,
  then $ActiveRequests \leftarrow ActiveRequests - 1$.
- If $AdmitNew = 0$, $ActiveRequests = A$, and $BlockedRequests = 0$,
  then $ActiveRequests \leftarrow ActiveRequests - 1$ and $AdmitNew = 1$, i.e., the admission control status changes and the system again starts accepting both types of requests: from new sessions and already accepted sessions.

– If $AdmitNew = 0$, $ActiveRequests = A$, and $0 < BlockedRequests \leq B$,
then one of the blocked requests is accepted for processing in the system and only the counter $BlockedRequests$ is updated: $BlockedRequests \leftarrow BlockedRequests - 1$.

We call this version of algorithm the *conservative AWAIT*. Under this algorithm the differentiation of requests from new and accepted sessions starts when the counter $ActiveRequests$ reaches its maximum value A. Then new sessions are rejected and requests from accepted sessions have extra buffering facility in the blocking queue. Once the $ActiveRequests$ counter gets below A, then the admission restriction is lifted and new session requests are again accepted.

We also introduce a different version of the algorithm, called *aggressive AWAIT*, which at a first glance is only slightly different from the *conservative AWAIT* above. However, the performance evaluation of these two versions shows a surprising difference in behavior and in the numbers of aborted and rejected sessions. As we see later, the *aggressive AWAIT* decreases forcefully the number of aborted sessions while supporting the same useful system throughput as the conservative *AWAIT*.

For the *aggressive AWAIT* strategy we introduce the additional variable $Overload$:

– *Overload* is a boolean variable that defines whether the system is under severe overload. Typically, $Overload = 0$ while the system can process all the requests from the already accepted sessions. $Overload = 1$ when system observes an aborted request from the accepted session. This may happen when $ActiveRequests = A$ and $BlockedRequests = B$, and the incoming request is from an accepted session. The aborted session triggers an "emergency situation" that is treated aggressively. New session requests are not accepted during overload until all the queues in the system are flushed. This helps in providing a prolonged preferential treatment of requests from the accepted sessions to rapidly overcome the overload state.

When the overload condition is triggered, i.e., $Overload = 1$, there are slightly different rules for updating the system state when a processed request leaves the system:

– If $AdmitNew = 0$, $Overload = 1$, $ActiveRequests = A$, and $BlockedRequests = 0$, then $ActiveRequests \leftarrow ActiveRequests - 1$, but the system is considered to be still under severe overload and its admission control status does not change: the system still rejects requests from new sessions and only processes requests from the already accepted sessions.
– If $Overload = 1$ and $ActiveRequests = 0$, then the operation of the system goes back to normal: $Overload = 0$ and $AdmitNew = 1$.

The pseudo-code shown in Figure 3 summarizes both versions of the *AWAIT* algorithm: conservative and aggressive. To unify the description, in the conservative version of the algorithm the state of variable $Overload$ does not change, i.e., $Overload = 0$.

In sum, the rationale for the *conservative* versus the *aggressive* version of the algorithm is the following. If the system operates under a burst, then queues tend to build up fast. An accepted session that is aborted signals the system about insufficient resource capacity for processing requests from already accepted sessions. To mitigate the performance effects of this, it is more effective to completely dedicate system resources for processing only the accepted session requests by flushing the system queues at the expense of a higher ratio of rejected new sessions. This strategy benefits accepted sessions by implicitly giving them priority and "reserving" the system for exclusive processing of accepted session requests (until overload subsides). In the following subsection, we present experimental evidence that shows the relative performance of the *conservative* versus the *aggressive* version of the algorithm.

```
For every request $req$ that arrives for processing
  if (AdmitNew AND ActiveRequests $<$ A)
    accept $req$
    ActiveRequests = ActiveRequests + 1
    if (ActiveRequests == A)
      AdmitNew = 0
  else if (!AdmitNew AND BlockedRequests $<$ B)
    if (type($req$) == NewSession)
      reject $req$
    if (type($req$) == AcceptedSession)
      accept $req$ into BlockingQueue
      BlockedRequests = BlockedRequests +1
  else if (!AdmitNew AND BlockedRequests == B)
    reject $req$      //Reject all requests
    if (type($req$)==AcceptedSession)  //Accepted session is aborted
      Overload=1    //Aggressive version: trigger overload state

For every request $req$ that leaves the system
  if (ActiveRequests $<$ A)
    ActiveRequests = ActiveRequests -1
  else if (ActiveRequests==A AND 0$<$BlockedRequests$\leq$B
    move one request from blocking queue to queue
    BlockedRequests = BlockedRequests -1
  else if (ActiveRequests == A AND BlockedRequests == 0)
    ActiveRequests = ActiveRequests -1
    if (!Overload)
      AdmitNew = 1
  if (ActiveRequests==0 ) //Aggressive version: queues flushed?
    Overload = 0    //Restore overload state to normal
    AdmitNew = 1    //Start admitting new sessions
```

Fig. 3. *AWAIT*: Admission control algorithm, aggressive version. The *conservative AWAIT* is obtained by removing the statements labeled **Aggressive version** .

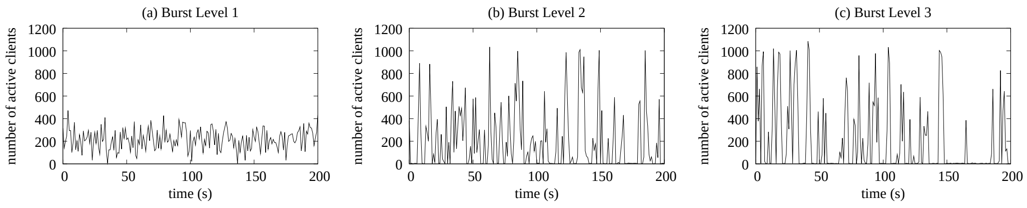

Fig. 4. The burstiness profiles of the three arrival MAPs.

3.2 Performance Evaluation of *AWAIT*

We evaluate the performance of *AWAIT* via trace driven simulation. Because our purpose is to evaluate the different proposed algorithms under varying burstiness levels, we conducted experiments assuming that arrivals of new sessions are bursty. We use a Markovian Arrival Process (MAP) to generate three arrival processes with the same mean and variance but with distinctive burstiness profiles. For details on the generation of the three MAP processes as well as on their effectiveness in mimicing bursty arrivals such those reported in the 1998 World Cup web server we direct the reader to [23]. The burstiness profiles (i.e., the number of arrivals as a function of time) for the three MAPs that we use for the arrival process are illustrated in Figure 4.

The service processes at the front server and the database server are also modeled via MAPs (see [22]) that accurate capture the service demands of TPC-W's ordering mix.[6] Each session consists of a sequence of requests that defines a *session length*. MAPs have been shown to be

[6] Experiments with TPC-W's ordering and browsing mixes were also conducted. Results are qualitatively the same as with the ordering mix and are not reported here due to lack of space.

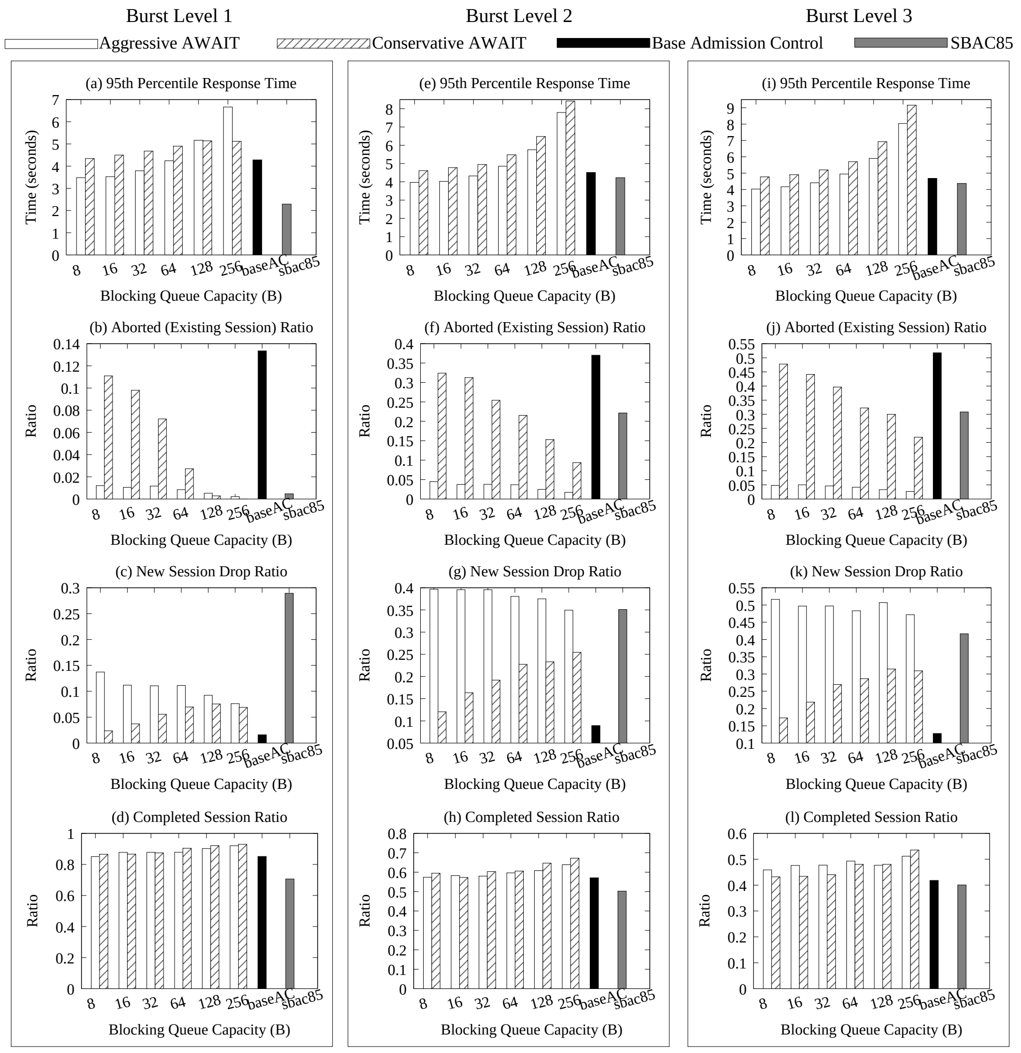

Fig. 5. *AWAIT* with fixed size of the blocking queue. The graphs illustrate performance values for the aggressive and conservative versions (see white and shaded bars, respectively) for various fixed sizes of the blocking queue B. In all experiments, the limit of accepted requests A is set to 256, based on capacity planning.

surprisingly compact yet very effective models of the service process in multi-tiered systems, modeling implicitly conditions such as caching or database locks (see [22]). Session lengths are uniformly distributed between parameters 5 and 35, that is with expected mean equal to 20.

Figure 5 illustrates the performance of the two versions of *AWAIT* as a function of the capacity of the blocking queue B. For reference, we also report on the performance of the system with simple admission control based on the number of *ActiveRequests* only (labeled: "baseAC") as well as the performance of SBAC with CPU utilization threshold set to 85%. Note that for *all* experiments, we set the *ActiveRequests* counter to 256, as suggested by the capacity planning study of Section 2.

The figure presents results for the three burstiness profiles in the arrivals of new sessions. First, one can easily see that the effect of the degree of burstiness in the arrivals dramatically impacts the user perceived performance, see the 95th percentiles of user response times for the various policies, first row of graphs. Looking just at the percentiles, it is clear that the addition of the blocking queue deteriorates the user end-to-end times but the real benefit of blocking can be seen in the decrease of the aborted session ratio, see the second row of graphs, as well as in the decrease of new session drop ratio, see the third row of graphs. The useful throughput of the system (measured in successfully completed sessions) is shown in the last row of graph that demonstrate the improved metric for both versions of *AWAIT* strategy compared to SBAC and *baseAC*.

Under low burstiness conditions, see first column of graphs, it is apparent that SBAC remains a good choice, at the expense of a very high percentage (nearly as high as 30%) of new session rejections. The aggressive and conservative versions of *AWAIT* result in longer response times but in significantly lower drop ratios, see Figures 5(b) and 5(c). With higher burstiness levels, the aggressive version results in better response time percentiles, see Figures 5(e) and 5(i).

The effectiveness of the aggressive version to keep the aborted session ratio low is apparent across all burstiness levels, see Figures 5(b), 5(f), and 5(j) (second row of graphs). These figures show that the aggressive version very effectively differentiates between existing and new sessions, and treats existing sessions preferentially.

Naturally, because of the limited system capacity, if the number of accepted sessions that are aborted is low, then the ratio of rejected new sessions is bound to increase. This effect is shown for the aggressive policy in the third row of graphs in Figure 5, but this is unavoidable since our purpose is to bias the system for processing the requests of already accepted sessions against admitting new sessions, especially under periods of bursty traffic.

However, intuitively, there is an additional concern on the effectiveness of the aggressive *AWAIT* strategy compared to its conservative version: “flushing” the system queues might result in a less efficient resource usage and potentially may lead to a lower useful throughput. The last row of graphs in Figure 5 answers this question. It shows that the useful throughput of the system measured in successfully completed sessions is very similar for both conservative and aggressive versions of *AWAIT* and also significantly higher than under earlier proposed SBAC strategy or the simple *baseAC* policy.

In summary, the Figures 5 shows that the aggressive *AWAIT* minimizes the number of aborted sessions while meeting service SLOs. Yet, its performance is sensitive to the capacity of the blocking queue B. In the following section we present an adaptive algorithm that changes the blocking queue capacity as a function of the workload burstiness in order to adaptively meet SLO targets.

3.3 Adaptive *AWAIT* Strategy

Here, we show how we can adjust on-the-fly the size of the blocking queue B in order to achieve a certain predefined SLO. Larger blocking queues result in longer user response times but have less aborted sessions.

To dynamically adjust the blocking queue size, we use historical information of the achieved 95th percentiles of all requests served by the system (irrespective of the blocking queue capacity used – this value should reflect the target system SLO as the size of the blocking queue is transparent to the user) but also response time percentiles that correspond to *every* other blocking queue capacity B used since the inception of the system. We use this information to decide whether the current blocking queue capacity is sufficient or not. Changing the blocking queue capacity B throughout the lifetime of the system is critical as during workload surges smaller

B's result in better performance rather than large B's.[7] To make readily available the values of the 95th percentiles of the user response times, we maintain for each blocking capacity B a corresponding histogram of the user response times for that B. Therefore, for each completed request, two response time histograms are updated: the histogram of all requests in the system (irrespective of the blocking capacity B) and the histogram that corresponds to the current block capacity B used.

We decide whether to change the capacity of the blocking queue for every group of $K = 10,000$ requests served.[8] The adaptive algorithm then compares the achieved response time percentiles of *all* jobs in the system and the response times percentiles of the current configuration B with the target SLOs. If both percentiles are less than the SLO and there are aborted sessions, then it is clear that we can reduce the aborted ratio because there is room to increase B (since response times percentiles do not violate the SLO). If both percentiles are greater than the SLO, then the blocking queue should be reduced in an effort to meet the SLO target. If none of the above two conditions are met, we opt to leave the blocking queue capacity in its current level, otherwise the system may suffer from thrashing. For example, if the response time percentile of all requests is violated, but the percentile of the current B is not, the algorithm still stays with the current blocking queue size B, since the system is on a positive state and its accumulated statistics eventually will correct the percentile of all requests.

The steps of increase/decrease of the blocking queue capacity can be arbitrary. In the experiments presented in this section, the capacity of the blocking queue B can have sizes as small as 1 and as large as 120. The increase/decrease step is equal to 5 for values of B less than 10 and equal to 20 for values of B greater than 20. We stress that other step values could also work, their selection may affect though how quickly the algorithm converges to a desirable B range. Figure 6 summarizes the algorithm.

```
For every aborted session
    AbortedSessions++
For every finished request
    counter++
    update total_RT_histogram (all requests, irrespective of $B$)
    update current_$B$_RT_ histogram (with current blocking queue $B$)
    if (counter == K)
      if (total_RT_percentile $<$ SLO AND current_$B$_RT_percentile $<$ SLO
      AND AbortedSessions $>$ 0)
          increase current blocking capacity $B$   // reduce aborted ratio
      if (total_RT_percentile $>$ SLO AND current_$B$_RT_percentile $>$ SLO)
          reduce current blocking capacity $B$   // meet SLO target
      counter = 0
      AbortedSessions = 0
```

Fig. 6. Policy for adapting the blocking queue size B in the enhanced, adaptive *AWAIT* strategy.

The effectiveness of the adaptive *AWAIT* strategy is illustrated in Figure 7. Here, we experimented with the three different burst levels but also using different target SLOs. The figure

[7] This may initially seem counter-intuitive as workload surges would result in large numbers of requests that are simultaneously in the system. However, in order to maintain the target SLOs during a surge it is necessary to limit the blocking queue capacity, otherwise the time spent there dominates user response times and SLOs are violated.

[8] We selected $K = 10,000$ to be able to collect meaningful statistics for a group of requests. K should be large enough for accumulating meaningful statistics, but different values, e.g., $K = 5,000$ or $K = 15,000$ will work too.

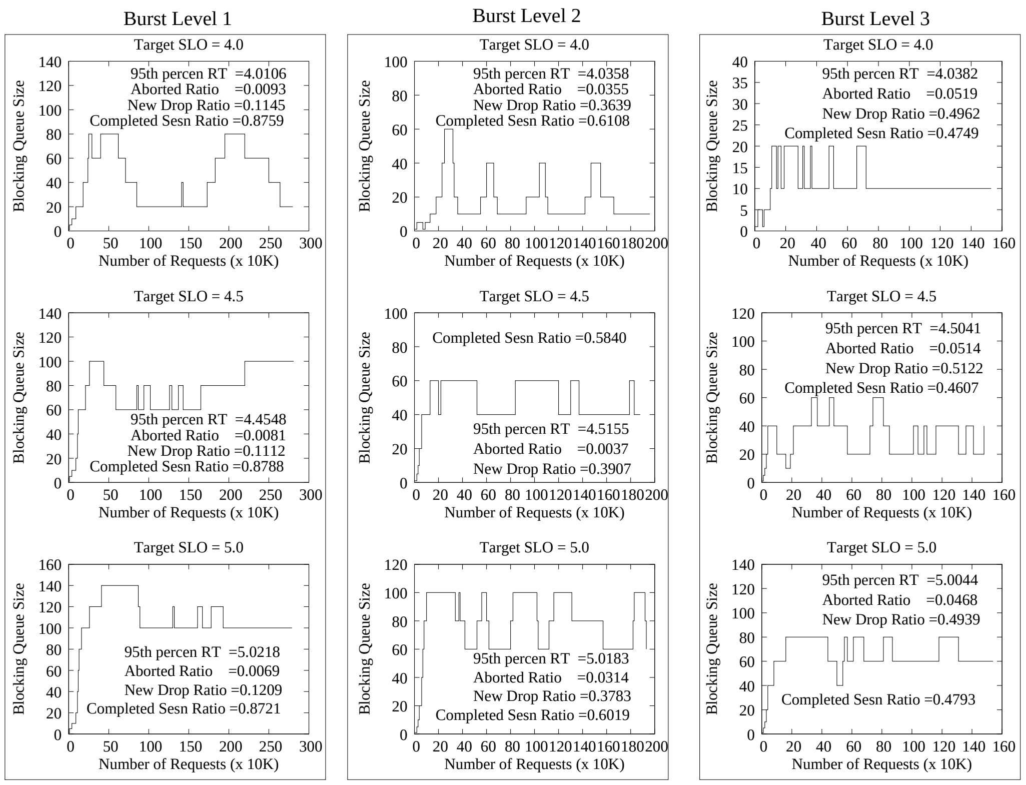

Fig. 7. Adaptive *AWAIT*: illustration of how the capacity of the blocking queue B changes as a function of the workload.

illustrates how the blocking queue size changes as a function of the number of requests that are processed by the system for the various experiments. In each graph we also report on the achieved 95th percentile of the round-trip time, as well as on the aborted and new session drop ratios. The figure shows that the adaptive *AWAIT* is remarkably robust: it reaches the target SLOs exceptionally well for all cases, while maintaining very low aborted rates. For each burst level, as the target SLO increases, the algorithm effectively increases the blocking queue capacity while reducing the aborted ratio. If we maintain the same SLO but change the burstiness of arrivals, the algorithm decreases the capacity of the blocking queue B. In all experiments, requests from existing sessions are preferentially treated as low aborted ratios across all experiments are reported, and the ratio of successfully completed sessions is higher under the adaptive *AWAIT* policy compared to the aggressive static *AWAIT* strategy introduced in Section 3.1. These results demonstrate the effectiveness and robustness of the proposed autonomic mechanism of the aggressive *AWAIT* policy.

4 Related Work

There has been a lot of research in the areas of overload control, service differentiation, request scheduling, and request distribution for Web servers and web server clusters. Due to space limitations, we provide a very brief overview here.

The use of admission control for an overload management has been proposed and explored in several systems. Iyer et al. [18] employ a simple admission control mechanism based on bounding the length of the Web server listen queue. The authors try minimizing the work spent on a request which is eventually not serviced due to overload. They analyze different queue management approaches and use multiple thresholds, though they do not specify how these thresholds should be set to meet a given performance target. Cherkasova and Phaal [13] introduce session-based admission control, driven by a CPU utilization threshold, which performs an admission decision based on user sessions rather than individual requests, and during the overload rejects new sessions while serving requests from already accepted sessions. Carlstrom and Rom [9] proposed a performance model for scheduling client requests and session-level admission control using generalized processor scheduling discipline. To improve the efficiency of session-based admission-control mechanisms and reduce its overhead, Voigt et al. [28, 29] present several kernel-level mechanisms for overload protection and service differentiation.

Many of the proposed techniques are based on fixed policies, such as bounding the maximum request rate of requests to some constant value. For example, PACERS [11] limits the number of admitted requests based on estimated web server capacity. The authors use a very simple simulated service where request processing time is linear function of the requested Web page size. Similar ideas (and similar problems with fixed threshold settings) are pursued in [8]. Web2K presents a mechanism prioritizing requests into two classes: premium and basic. Connection requests are forwarded into two different request queues, and admission control is performed using two metrics: the accept queue length and measurement-based predictions of arrival and service rates from that class. Bartolini et al., in their recent work [6, 7], introduce a quite elaborate session admission algorithm, called AACA, that self-configures a dynamic constraint on the rate of incoming new sessions to satisfy guarantees of the Service Level Agreements (SLA). However, the rate limitation for the next iteration interval is based on a relatively straightforward prediction of the session arrival rate from the previous interval measurements.

Many earlier papers combine differentiated service with admission control [3, 15, 19, 20, 28]. Kanodia and Knightly [19] develop an admission control and service differentiation mechanism which is based on a general framework of request and service envelopes. Such envelopes statistically describe the server's request load and service capacity as a function of interval length.The proposed mechanism integrates latency targets with admission control and improves the percentage of requests that meet their QoS delay requirements. The approach is evaluated via a trace-driven simulation.

A number of systems have explored a controlled content adaptation [1, 10, 17] for scaling web site performance, i.e., degrading the quality of static Web content by reducing the resolution and the number of images delivered to clients.

Several research papers have examined how control theory can be applied in the context of Web servers [2, 21, 24]. Lu et al. [21] present a control-theoretic approach to provide guaranteed relative delays between different service classes. Main challenge in such works is that good models of system behavior are difficult to derive. In particular, the Internet applications are subject to poorly understood traffic and resource demands. The papers mentioned here make use of linear models, which may be inaccurate in describing systems with bursty loads and resource requirements.

Many earlier papers study request and connection scheduling for improving Web server performance [12, 14, 16]. While shortest job first scheduling for static content Web sites can improve performance of a web server, it can not prevent it from overload though. Elnikety et al. [16] present an elegant solution for admission control and request scheduling for multi-tier e-commerce sites,. Their method is based on measuring the execution costs of requests online, distinguishing different request types, and performing both overload protection and preferential scheduling using a straightforward control mechanism. They implement their admission control using proxy, called Gatekeeper, with standard software components on the Linux operating system. There were a few other works close to Gatekeeper in spirit, SEDA [31] is a prime example of such work. In SEDA, applications consist of a network of event-driven stages connected by explicit queues. SEDA makes use of a set of dynamic resource controllers by preventing resources from being over-committed when demand exceeds service capacity. It keeps stages within their operating regime despite large fluctuations in load and allows services to be well-conditioned to load, i.e., preventing their performance degradation under severe overload. The authors describe several control mechanisms for automatic tuning and load conditioning, including thread pool sizing, event batching, and adaptive load shedding.

5 Conclusions

We presented an autonomic policy for service differentiation and admission control during overload for multi-tiered system management that offer web services. We focused on the pitfalls of existing policies under bursty conditions and remedy the problem by proposing the concept of a blocking queue where requests from already accepted sessions are stored if the system operates in overload. This blocking queue benefits performance by minimizing the dropped requests of already accepted sessions but also contributes to the end-to-end user perceived system response time. We proposed a novel autonomic algorithm, called *AWAIT*, that can limit the increase of the end-to-end response times within predefined SLO targets while dynamically adjusting the capacity of the blocking queue to the workload burstiness. Detailed simulations with the widely used TPC-W e-commerce benchmark under a variety of workload burstiness levels support the effectiveness and robustness of *AWAIT*.

The current algorithm adapts the blocking queue capacity to shield the offered web service from bursty arrivals, to provide service differentiation, and to prevent the system from overload. It complements the basic overload mechanism that sets a limit on the number of active client requests that are simultaneously processed by the system. Currently, this limit is defined by capacity planning. In our future work, we plan to automate the capacity planning step as well, i.e., to adjusts the value of this basic parameter on-the-fly when the workload profile experiences significant changes. We are also working on theoretically determining the ideal blocking queue capacity given a level of workload burstiness.

References

1. T. Abdelzaher and N. Bhatti. Web content adaptation to improve server overload behavior. Computer Networks, 31(11-16), 1999.
2. T. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. IEEE Transactions on Parallel and Distributed Systems, 13(1), 2002.
3. J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in Web content hosting. In Workshop on Internet Server Performance, Madison, WI, June 1998.
4. Apache Software Foundation. The Apache Web server. http://www.apache.org
5. S. Balsamo, V. de Nitto Personé, R. Onvural. Analysis of Queueing Networks with Blocking, Kluwer Academic Publishers, 2001.

6. N. Bartolini, G.Bongiovanni, S.Silvestri, An autonomic admission control policy for distributed web systems. Proc. of the Intl. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'07), 2007.
7. N. Bartolini, G. Bongiovanni, and S. Silvestri. Self-* overload control for distributed web systems. Proc. of the Intl. Workshop on Quality of Service (IWQoS), 2008.
8. P. Bhoj, S. Ramanathan, and S. Singhal. Web2K: Bringing QoS to Web servers. Technical Report HPL-2000-61, HP Labs, May 2000.
9. J. Carlstrom and R. Rom. Application aware admission control and scheduling in web servers. Proc. of INFOCOM, 2002.
10. S. Chandra, C. Ellis, and A. Vahdat. Differentiated multimedia Web services using quality aware transcoding. Proc. of INFOCOM, 2000.
11. X. Chen, P. Mohapatra, and H. Chen. An admission control scheme for predictable server response time for Web accesses. Proc. of the 10th World Wide Web Conference (WWW), Hong Kong, 2001.
12. L. Cherkasova. Scheduling strategy to improve response time for Web applications. Proc. of High Performance Computing and Networking (HPCN), Amsterdam, April 1998.
13. L. Cherkasova and P. Phaal. Session-based admission control: A mechanism for peak load management of commercial Web sites. IEEE Transactions on Computers, 51(6), June 2002.
14. M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in Web servers. Proc. of the USENIX Symposium on Internet Technologies and Systems (USITS), 1999.
15. L. Eggert and J. Heidemann. Application-level differentiated services for Web servers. World-Wide Web Journal, 2(3), Aug, 1999.
16. S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. Proc. of the World Wide Web Conference (WWW), 2004.
17. A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP), 1997.
18. R. Iyer, V. Tewari, and K. Kant. Overload control mechanisms for Web servers. In Workshop on Performance and QoS of Next Generation Networks, Nagoya, Japan, November 2000.
19. V. Kanodia and E. W. Knightly. Ensuring latency targets in multiclass Web servers. IEEE Transactions on Parallel and Distributed Systems, 13(10), October 2002.
20. K. Li and S. Jamin. A measurement-based admission-controlled Web server. Proc of INFOCOM, 2000.
21. C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A feedback control approach for guaranteeing relative delays in Web servers. In IEEE Real-Time Technology and Applications Symposium, 2001.
22. N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Burstiness in multi-tier applications: Symptoms, causes, and new models. In ACM/IFIP/USENIX International Middleware'08, 2008.
23. N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting realistic burstiness to a traditional client-server benchmark. Proc. of the 6th Intl. Conference on Autonomic Computing (ICAC), 2009.
24. S. Parekh, N. Gandhi, J. L. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. Proc. of the IFIP/IEEE International Symposium on Integrated Network Management, 2001.
25. H.G. Perros. Queueing networks with blocking. Oxford University Press, 1994.
26. K. Shen, H. Tang, T. Yang, and L. Chu. Integrated resource management for cluster-based Internet services. Proc.of Operating Systems Design and Implementation (OSDI), 2002.
27. TPC-W Benchmark. URL `http://www.tpc.org`
28. T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel mechanisms for service differentiation in overloaded Web servers. Proc. of the USENIX Annual Technical Conference, 2001.
29. T. Voigt. Overload Behaviour and Protection of Event-driven Web Servers. Proc. of International Workshop on Web Engineering, 2002.
30. M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. Proc. of the USENIX Symposium on Internet Technologies and Systems (USITS), 2003.
31. M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. Proc. of the 18th Symposium on Operating Systems Principles (SOSP), 2001.