HEWLETT PACKARD

Hybrid and Predictive Admission Strategies to Improve the Performance of an Overloaded Web Server

Ludmila Cherkasova, Peter Phaal
Computer Systems Laboratory
HP Laboratories Palo Alto
HPL-98-125(R.1)
May, 1999

E-mail: [cherkasova,phaal]@hpl.hp.com

web servers, overloaded conditions, admission control strategy, performance analysis, optimization, SBAC

In this paper, we use a session-based workload to measure a web server's performance. We define a session as a sequence of client requests. An overloaded web server can experience a severe loss of throughput when measured as the number of completed sessions. Moreover, the overloaded web server discriminates against longer sessions. This could significantly impact sales and profitability of commercial web sites because longer sessions are typically the ones that would result in purchases.

Session based admission control (SBAC), introduced in [CP98], prevents a web server from becoming overloaded and ensures that longer sessions can be completed. If a server is functioning near its capacity a new session will be rejected (or redirected to another server if one is available). If there is enough capacity, the admission control mechanism will admit a new session and process all future requests related to it.

In this paper, we propose two new admission control strategies: hybrid and predictive, aiming to optimize the performance of SBAC mechanism. The hybrid strategy is based on a self-tunable admission control function, adjusting itself accordingly to variations in traffic loads. It shows improved performance results for workloads with medium to long average session length. The predictive strategy estimates the number of new sessions it can accept and still guarantee processing of all future session requests. It consistently shows the best performance results for different workloads and different traffic loads.

Internal Accession Date Only

Contents

1 Introduction

As complex business applications are deployed on the web it is becoming very difficult to ensure adequate level of services to customers who are becoming increasingly reliant on the services.

Commercial applications impose a set of additional, service level expectations. Typically, access to a web service occurs in the form of a *session* consisting of many individual requests. Placing an order through the web site involves further requests relating to selecting a product, providing shipping information, arranging payment agreement and finally receiving a confirmation. So, 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.

In this paper, we continue studying web server behaviour and performance using session-based workload. Web server performance, measured in completed sessions versus measured in processed requests, becomes very different for loads that exceed server capacity.

If a load consists of single, unrelated requests then the server throughput is defined by its maximum capacity, i.e. a maximum number of connections the server can support. Any extra connections will be refused. Thus, once a server has reached its maximum throughput, it will stay there, at a server maximum capacity.

However, if the server runs a session-based workload then a dropped request could occur anywhere in the session. That leads to aborted, incomplete sessions. We showed in [CP98] that an overloaded web server can experience a severe loss of throughput when measured in completed sessions. As an extreme, a web server which seems to be busily satisfying clients requests and working at the edge of its capacity could be wasting its resources on failed sessions and, in fact, not be accomplishing any useful work. Statistical analysis of completed sessions reveals that an overloaded web server discriminates against the longer sessions. Our analysis of a retail web site showed that sessions resulting in sales are typically 2-3 times longer than non-sale sessions. Hence discriminating against the longer sessions could significantly impact sales and profitability of the commercial web sites.

The main goal of a session based admission control, introduced in [CP98], is the prevention of web server overload. An admission control mechanism will accept a new session only when a server has the capacity to process all future requests related to the session, i.e. a server can guarantee the successful session completion. If a server is functioning near its capacity, a new session will be rejected (or redirected to another server if one is available).

In [CP98], we introduced a simple implementation of session based admission control based on server CPU utilization. We examined trade off between two desirable properties for an admission control mechanism: *responsiveness* and *stability* and introduced a family of admis-

sion control policies (ac-policies) which cover the space between ac-stable and ac-responsive policies.

If a server's load during previous time intervals is consistently high, and exceeds its capacity, then responsiveness is very important: the admission control policy should be switched on as soon as possible, to control and reject newly arriving traffic. However, if the server receives an occasional burst of new traffic, while still being under a manageable load, then the stability, which takes into account some load history, is a desirable property. It helps to maximize server throughput and does not unnecessary reject newly arriving traffic.

As we can see, these two properties are somewhat contradictory:

- responsiveness leads to a more restrictive admission policy (since there is a chance of "over reacting" to occasional traffic bursts while overall a server is not yet overloaded). It aims to minimize the number of aborted sessions and to achieve higher levels of service, at a price of slightly lower server session throughput (in particular, when a server operates in a heavy load area but is not yet overloaded).
- stability takes into account a server's load history. In such a way, that it delays the first reaction of the admission control policy to the overload, while it still looks like an occasional burst, rather than a consistent overload. If a total server load is still around the server capacity then such a strategy allows better server session throughput to be achieved. However, if the overload is consistent then a less restrictive rejection policy inevitably leads to a higher rate of aborted sessions, and as result to poorer session completion characteristics.

Obviously, a *hybrid* admission control strategy is a desirable goal. In this paper, we design such a *hybrid* strategy and analyze its performance. We show that proposed hybrid strategy successfully combines most attractive features of both ac-responsive and ac-stable policies. It shows improved performance results for workloads with medium to long average session length.

We analyzed why workloads with short average session lengths are most difficult to manage. We design a new, *predictive* admission control strategy which estimates the number of new sessions a server can accept and still guarantee processing of all future session requests. This strategy consistently shows the best performance results for different workloads and different traffic patterns. For workloads with short average session length, predictive strategy is the only strategy which provides both: highest server throughput in completed sessions and no (or, practically no) aborted sessions.

Proposed hybrid and predictive admission policies allows the design of a powerful admission control mechanism which tunes and adjusts itself for better performance across different workload types and different traffic loads.

The remainder of the paper presents our results in more detail. Section 2 introduce a new model of workload based on sessions. Section 3 outlines the structure and basic features of the web server model. Both sections discuss and extract the essential server and client parameters necessary to build a simplified simulation model. Section 4 introduces session based admission control and explains its main properties as well as related performance benefits. It defines a family of ac-policies ranging from ac-stable to ac-responsive. Section 5 introduces a new admission control strategy, called *hybrid*, Section 6 discuss disadvantages of server CPU utilization based implementation of SBAC. Section 7 introduces a new, *predictive* admission control strategy. Section 8 provides the performance comparison of different admission control strategies.

In order to introduce two new admission control strategies, we have to repeat some of the definitions and results related to session based admission control shown in [CP98]. Those who read this paper could directly go to Sections 5, 6, 7, 8.

2 Workload Model: Requests and Sessions

WebStone [WebStone] and SpecWeb96 [SpecWeb96] are the industry standard benchmarks for measuring web server performance. Using a finite number of clients to generate HTTP requests they retrieve different length files according to a particular file size distribution.

For example, SpecWeb96 file mix is defined by the files (requests) distribution from the following four classes:

- 0 Class: 100bytes - 900bytes (35%)
- 1 Class: 1Kbytes - 9Kbytes (50%)
- 2 Class: 10Kbytes - 90Kbytes (14%)
- 3 Class: 100Kbytes - 900Kbytes (1%)

The web server performance is measured as a maximum achievable number of connection per second supported by a server when retrieving files in the required file mix.

Commercial applications exhibit very different behavior: a typical access to a web service consists of a sequence of steps (a sequence of individual requests). A transaction is successful only when the whole sequence of requests is completed. The real measure of server performance is the server's ability to process the entire sequence of requests needed to complete a transaction.

We introduce a notion of a session as a unit of session workload. *Session* is a sequence of clients individual requests.

In our simulation, the session structure is defined by the following parameters:

- client (sender) address;
- original session length;
- current session length;
- time stamp when the session was initiated.

For a new session, the original and current session lengths coincide. For a session in progress, the current session length reflects the number of requests left to complete.

We define a request as a structure specified by the following parameters:

- the session that originated the request;
- requested file size;
- time stamp when the request was issued.

Throughout this paper, we consider a file mix as defined by a SpecWeb96. So, the individual requests retrieve the files defined by a SpecWeb96 distribution.

The client issues the next request only when it receives a reply for the previous request. The client issues its next request with some time delay, called *think time*. Think time is a part of the client definition rather than a session structure. The client waits for a reply for a certain time, called *timeout*. After a timeout, the client may decide to repeat its request – this is termed a retry. A limit is placed on retries – if this limit is reached and the reply is not received in time, both the request and the whole session is aborted.

Thus, a client model is defined by the following parameters:

- client address;
- think time between the requests of the same session;
- timeout - a time interval where the client waits for a server reply before reissuing the request;
- the number of retries before the session is aborted.

A session is successfully completed when all its requests are successfully completed. We will evaluate web server performance in terms of successfully completed sessions.

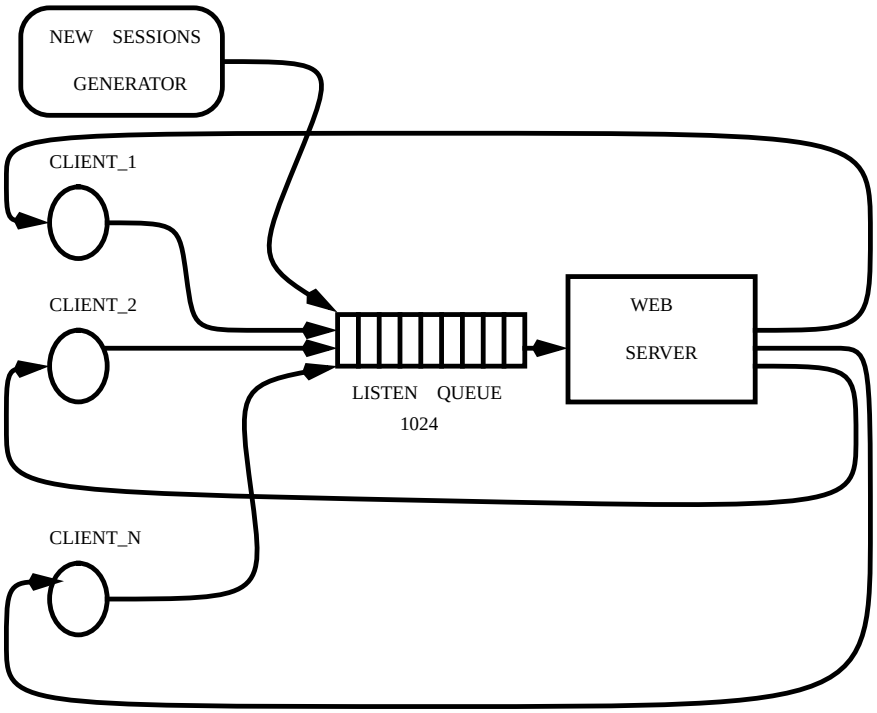

Figure 1: Basic Structure of Simulation Model.

For the rest of the paper, we assume the session lengths to be exponentially distributed with a given mean. In order to analyze the server behavior depending on a session length, we have performed experiments for session lengths with a mean of 5, 15 and 50.

3 Server Model: Functionality and Basic Parameters

To understand the difference in web server behavior while it runs request-based or session-based workloads we built a simulation model using C++Sim [Schwetman95]. Basic structure of the model is outlined in Figure 1.

It consists of the following components:

- a session workload generator;
- N clients;
- a web server.

A session workload generator produces a new session request accordingly to specified input model parameters:

- session load and

- sessions length distribution.

A session request (i.e first request of a session) is sent to a web server and is stored in the server *listen queue*. We limit the size of the listen queue to 1024 entries which is a typical default value. In this way, we are able to use an open model for sessions generation. Each consequent request from a session is issued and handled by a specified client. Client behavior is defined by a closed (feed back) loop model: the client issues the next session request only when it receives a reply from the previous request.

Two reasons could cause a request, and a session it belongs to, to be aborted:

- if a listen queue is full then the connection to a server is refused, and both the request and the whole session is aborted.
- after issuing the request, the client waits for a server reply for a certain time. After timeout, the client resends the request. There is limited number of *retries*. If the reply still has not been received in time, both the request and a whole session is aborted.

REMARK: When the client receives "connection refused" message due to a full listen queue, he/she can try to resend the request again. In case of overloaded server, it only can worsen the situation. We decided to simplify the model by aborting the request and the whole session, when a listen queue is full, without an additional client retry.

In previous paper [CP98], we have analized and justified the choice of model parameters to narrow the simulation space. Without loss of generality for the rest of the paper, we assume a model with the following client parameters:

- a think time between the requests of the same session is exponentially distributed with a mean of 5 seconds;
- a timeout - the time client waits for a reply before resending the request - is set to 1 second;
- a number of retries to resend the request after timeout is 1.

We also assume that a server capacity is 1000 connections per second for SpecWeb96 (current typical web servers running SpecWeb96 achieve 1000 - 4000 connections per second per processor). This assumption does not influence the results validity. In those rare cases, when assumed server speed can influence the results of the study, we will have special remarks and discussion related to the matter.

4 Session Based Admission Control Mechanism: Responsiveness vs Stability

The main goal of an admission control mechanism is to prevent a web server from becoming overloaded. We introduce a simple admission control mechanism based on the server CPU utilization.

The basic idea of a session based admission controller is as follows: the server utilization is measured during predefined time intervals (say, each second). Using this measured utilization (for the last interval) and some data characterizing server utilization in the resent past, it computes an "observed" utilization. If the observed utilization gets above a specified threshold then for the next time interval (i.e. the next second), the admission controller will reject all the new sessions and will only serve the requests from already admitted sessions. Once the observed utilization drops below the given threshold, the server (controller) changes its policy for the next time interval and begins to admit and process new sessions again.

Formally, the admission control mechanism is defined by the following parameters:

- U_{ac} – an *ac-threshold* which establishes the critical server utilization level to switch on the admission control policy;
- $T_1, T_2, ..., T_i, ...$ – a sequence of time intervals used for making a decision whether to admit (or to reject) new sessions during the next time interval. This sequence is defined by the *ac-interval length*;
- f_{ac} – an *ac-function* used to evaluate the observed utilization.

We will distinguish two different values for server utilization:

- $U_i^{measured}$ – a measured server utilization during T_i – the i-th ac-interval;
- $U_{i+1}^{observed}$ – an observed utilization computed using a given ac-function f_{ac} after ac-interval T_i and before a new ac-interval T_{i+1} begins, i.e. $U_{i+1}^{observed} = f_{ac}(i+1)$.

In this paper, we will consider ac-function $f_{ac}(i+1)$ defining $U_{i+1}^{observed}$ in the following way:

- $f_{ac}(1) = U_{ac}$;
- $f_{ac}(i+1) = (1-k) * f_{ac}(i) + k * U_i^{measured}$, where k is a damping coefficient between 0 and 1, and it is called *ac-weight coefficient*.

A web server with an admission control mechanism re-evaluates its admission strategy on specified by the time intervals $T_1, T_2, ..., T_i, ...$ boundaries. Web server behavior for the next time interval T_{i+1} is defined in the following way:

- If $U_{i+1}^{observed} > U_{ac}$ then any new session arrived during T_{i+1} will be rejected, and web server will process only requests belonging to already accepted sessions.
- If $U_{i+1}^{observed} \leq U_{ac}$ then web server during T_{i+1} is functioning in a usual mode: processing requests from both new and already accepted sessions.

Session based admission control combines several functions:

- it measures and observes the server utilization;
- it rejects new sessions when the server becomes critically loaded;
- it sends an explicit message of rejection to the client of a rejected session.

We believe that sending a clear message of rejection to a client is very important. It will stop clients from unnecessary retries which could only worsen the situation and increase the load on the server. If the server promises to serve these clients, say in five minutes, it might be enough to resolve the current overload and provide a high level of service without loosing customers. In our simulation model, we assume that processing a session rejection is equivalent to processing an average size file, that is similar to issue an explicit rejection message.

There are two desirable properties for an admission control mechanism: *responsiveness* and *stability*.

If a server's load during previous time intervals is consistently high, and exceeds its capacity, then "fast reaction", responsiveness is very important: the admission control policy should be switched on as soon as possible, to control and reject newly arriving traffic.

However, if the server receives an occasional burst of new traffic, while still being under a manageable load, then the "slow reaction", stable admission control policy, which takes into account some load history, is a desirable property. It helps to maximize server throughput and does not unnecessary reject newly arriving traffic.

As we can see, these two properties are somewhat contradictory:

- responsiveness leads to a more restrictive admission policy. It aims to minimize a number of aborted sessions and to achieve higher levels of service at a price of slightly lower server session throughput (in particular, when a server operates in a heavy load area but is not yet overloaded).

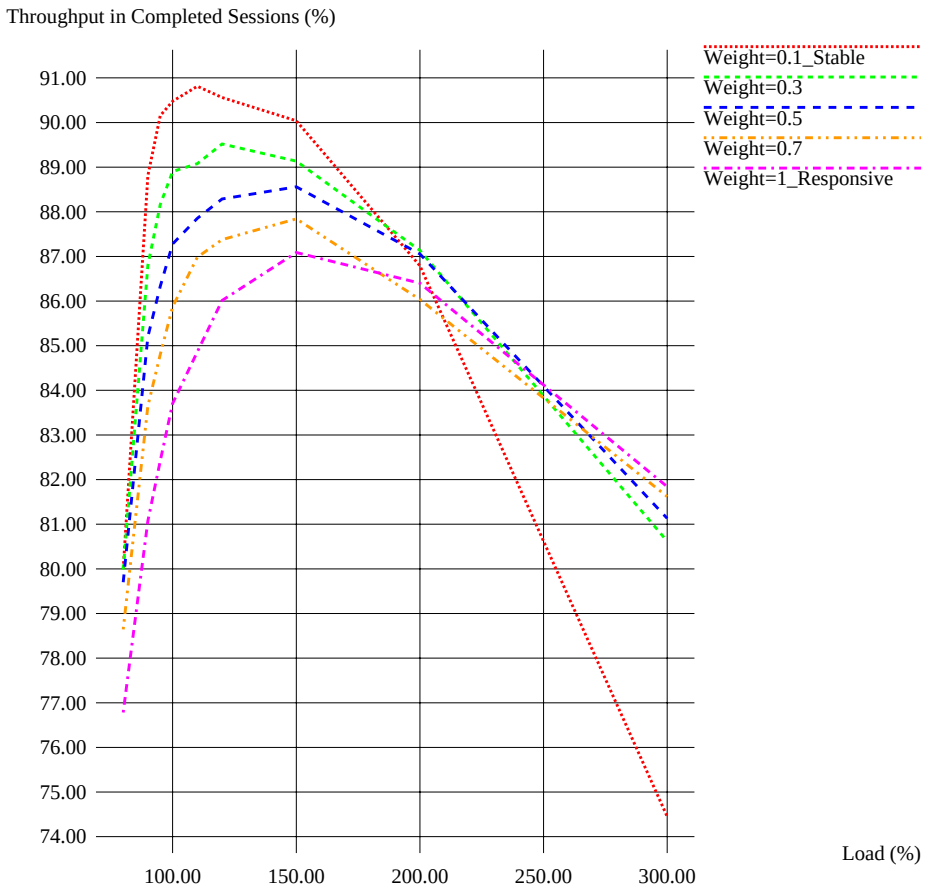

Figure 2: Throughput in Completed Sessions for Family of AC-Functions: from AC-Stable to AC-Responsive, Workload with Average Session Length of 15.

- stability takes into account a server's load history. In such a way, that it delays a first reaction of an admission control policy to the overload. If a total server load is still around the server capacity then such a strategy allows better server session throughput to be achieved. However, a less restrictive rejection policy inevitably leads to a higher rate of aborted sessions, and as result to poorer session completion characteristics.

The value of coefficient k in definition of f_{ac} introduces a family of admission control policies which cover the space between ac-stable and ac-responsive policies. If $k = 1$ then the admission control policy is based entirely on a value of measured server utilization during the last ac-interval. Let us call this strategy *ac-responsive*. If $k = 0.1$ then the admission control policy decision is strongly influenced by a server load prehistory, while the impact of a measured server utilization during the last ac-interval is limited. Let us call this strategy *ac-stable*.

Figure 2 shows the server throughput while running workload with average session length of 15, depending on ac-weight k used in ac-function f_{ac} definition.

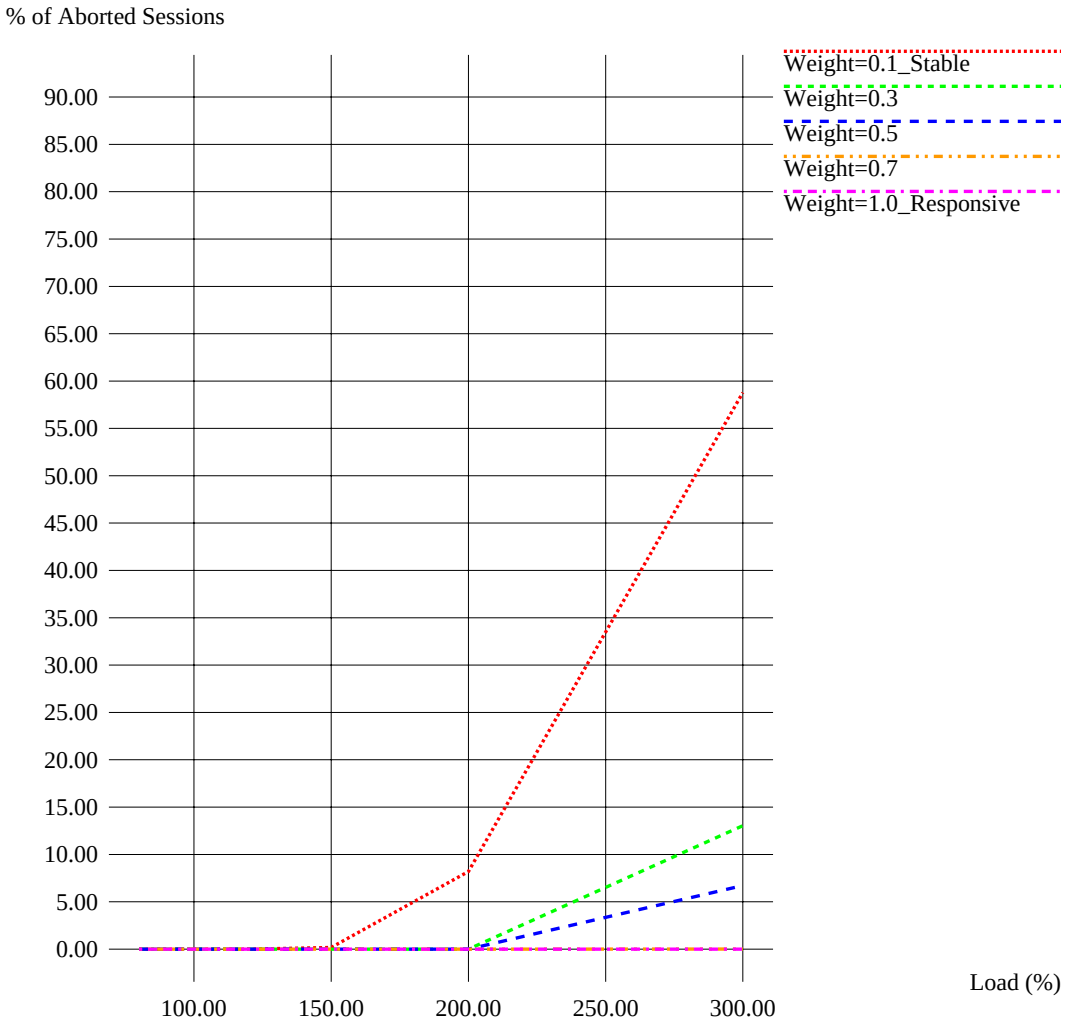

Figure 3: Number of Aborted Sessions for Family of AC-Functions: from AC-Stable to AC-Responsive, Workload with Average Session Length of 15.

As expected, the server throughput is higher under "more stable" ac-functions for a load below 170%. The situation changes for higher load in favor of "more responsive functions". The rates of aborted sessions are worse for "more stable functions" in higher load area as shown in Figure 3.

This shows again that ac-stable admission control functions achieve better throughput in the load range of 85%-120% at a price of higher number of aborted sessions under higher loads. While ac-responsive admission control functions lead to more restrictive admission policies and achieve better levels of service, especially at high loads but at the price of slightly lower server session throughput (in particular, when a server operates at loads in the range 85%-120%).

Obviously, a hybrid admission control strategy is a desirable goal.

5 Hybrid Admission Control Strategy

Once a web server is augmented with an admission control mechanism, the following question arrises: how to measure the "goodness" and efficiency of this mechanism in practice?

The following two values help to reflect an admission control "goodness":

- first of all, the percentage of aborted requests, which server can determine based on the client side closed connections. Aborted requests indicate that the level of service is unsatisfactory. Typically, aborted requests lead to aborted sessions, and could serve as a good warning sign of degrading server performance;
- second, a percentage of "connection refused" messages sent by a server, in the case of full listen queue. Refused connections are the dangerous warning sign of an overloaded server and its inevitable poor session performance.

If both of these values are zero then it reflects that an admission control mechanism uses an adequate ac-function f_{ac} to cope with current workload and traffic rate. Occurrences of aborted requests or refused connections reflect that ac-function f_{ac} has to be more responsive.

From the other side, if percentage of aborted requests and refused connections is zero, it could be also the case, that ac-function f_{ac} is too restrictive and hence, server is rejecting some of the sessions which it can handle otherwise.

We use these observations to design a self-tunable admission control strategy, called *hybrid*.

The idea is very natural: once some aborted requests or refused connections are observed, the admission strategy is adjusted to be ac-responsive, which is the most restrictive admission policy. After that, this strategy is kept unchanged for a time long enough to observe an "average session life". If during this time interval there are no aborted requests or refused connections then strategy is adjusted to be "slightly less responsive". In such a way, the admission strategy trying to migrate closer to ac-stable strategy until occurrences of aborted requests or refused connections signal the necessity to switch to ac-responsive strategy. There is some similarity between this idea and the method, the internet protocol TCP-IP uses, to adjust itself in presence of congestion.

In the following, more formal strategy description, we heavily use denotations and terminology introduced in Section 4.

Let $Ab(i)$ denote a number of aborted requests and refused connections accumulated during the time interval T_i.

Let *ac-cycle* define a number of time intervals, we will observe f_{ac} for aborted requests and

refused connections before we adjust ac-function to be less responsive. In the simulation model, it is defined by a $ThinkTime \times SesLength$ that is an approximation of an average session "life". This time interval aims to reflect a cycle (from the sessions admission to their completion) of SBAC working with a new, adjusted ac-function, and it is estimated to be long enough to evaluate "goodness" of this function. We will discuss later, in the Section 7, how to approximate *ac-cycle* in practice.

Hybrid strategy adjusts its admission function f_{ac} in the following two situations:

- Let $Ab(i) > 0$ during the time interval T_i. Then for the next time interval T_{i+1} the ac-weight k in ac-function f_{ac} is adjusted to 1 (i.e. $k = 1$), changing f_{ac} to become the ac-responsive function with most restrictive admission policy.
- Let since the last time admission function was adjusted, and for the whole duration of the *ac-cycle*, the number of the aborted requests and refused connections stay equal to zero:

$$\sum_{j=i+1}^{i+1+ac\_cycle} Ab(j) = 0.$$

  Then for the next time interval T_{n+1} (where $n = i + 1 + ac\_cycle$) the ac-weight k in ac-function f_{ac} is decreased by 0.1 (i.e.$k = k - 0.1$), changing f_{ac} to become slightly less "responsive" ac-function with less restrictive admission policy.

Two steps, described above, repeat depending on situation. If for the next *ac-cycle* the number of aborted requests and refused connections is zero, then ac-weight k in ac-function f_{ac} is decreased further by 0.1 ($k = k - 0.1$), and it continues to adjust in similar way, until it becomes ac-stable policy. Otherwise, if for some time interval T_j during the *ac-cycle* $Ab(j) > 0$ then, as it was described before, ac-weight k is adjusted to 1 again, changing f_{ac} back to ac-responsive function.

In such a way, f_{ac} adjusts itself between ac-stable and ac-responsive policies accordingly to traffic rate requirements.

REMARK1. This idea of adjusting the policy parameters in reaction to aborted requests and refused connections can be taken further: to adjust an admission policy threshold U_{ac} to a correct level.

REMARK2. In our simulation model, aborted requests indicate that the level of service is unsatisfactory. In real life, certain percent of aborted, "no-reason-why" requests is always present. Hybrid strategy should be defined to react when a percentage of aborted requests above "no-reason-why" one.

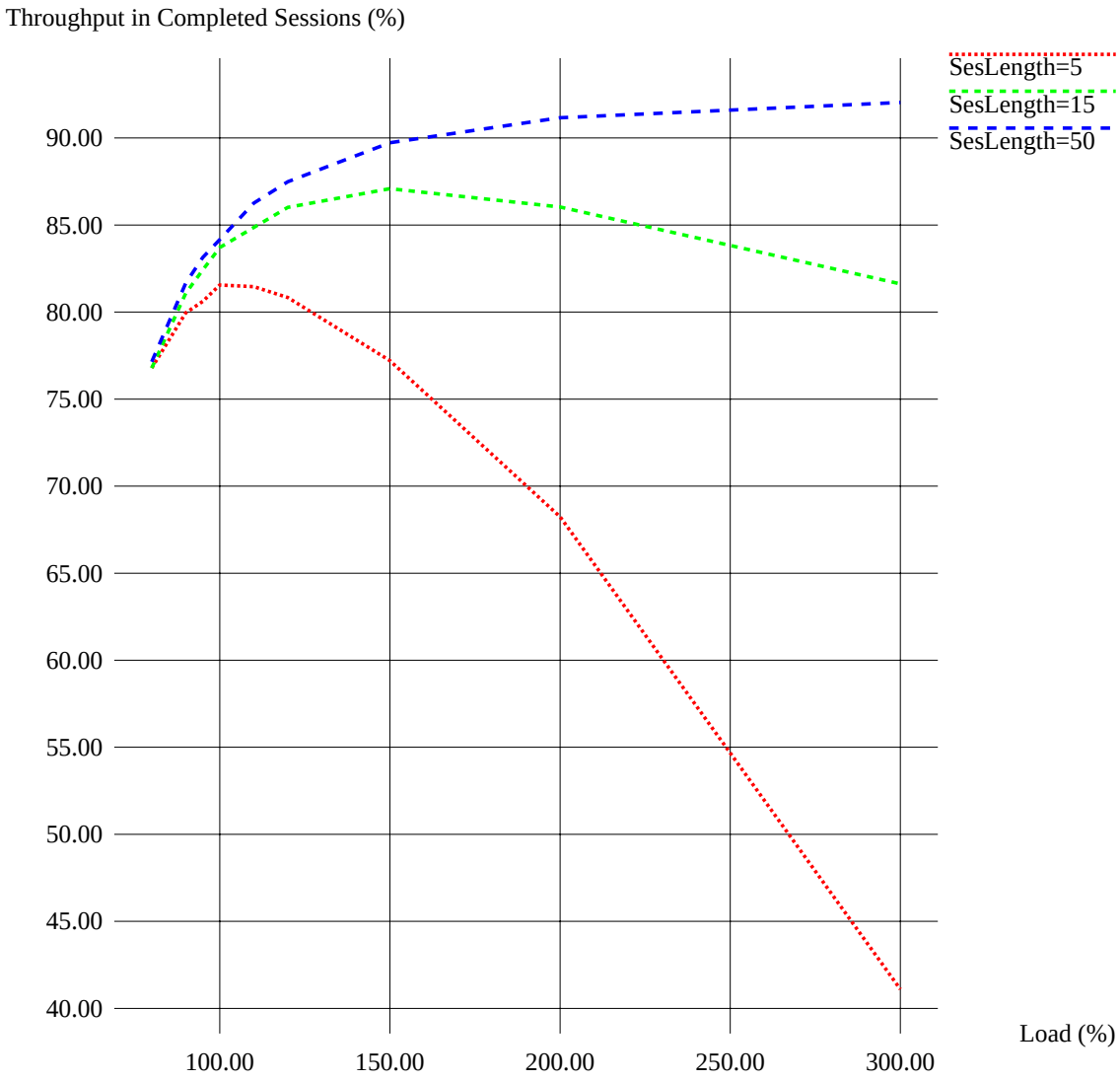

Figure 4: Throughput in Completed Sessions for Server with Admission Control.

6 Disadvantages of CPU Utilization Based Implementation of SBAC

In [CP98], we analyzed simulation results for a server augmented with *SBAC*, using ac-responsive strategy and ac-threshold $U_{ac} = 95\%$, for the average session lengths of 5, 15 and 50. We have varied a load from 80% to 300%.

Figure 4 shows server throughput in completed sessions.

One of the goals of the admission control mechanism is to minimize the number of aborted sessions (ideally, reducing them to 0) by explicit session rejection. Figure 5 shows the per-

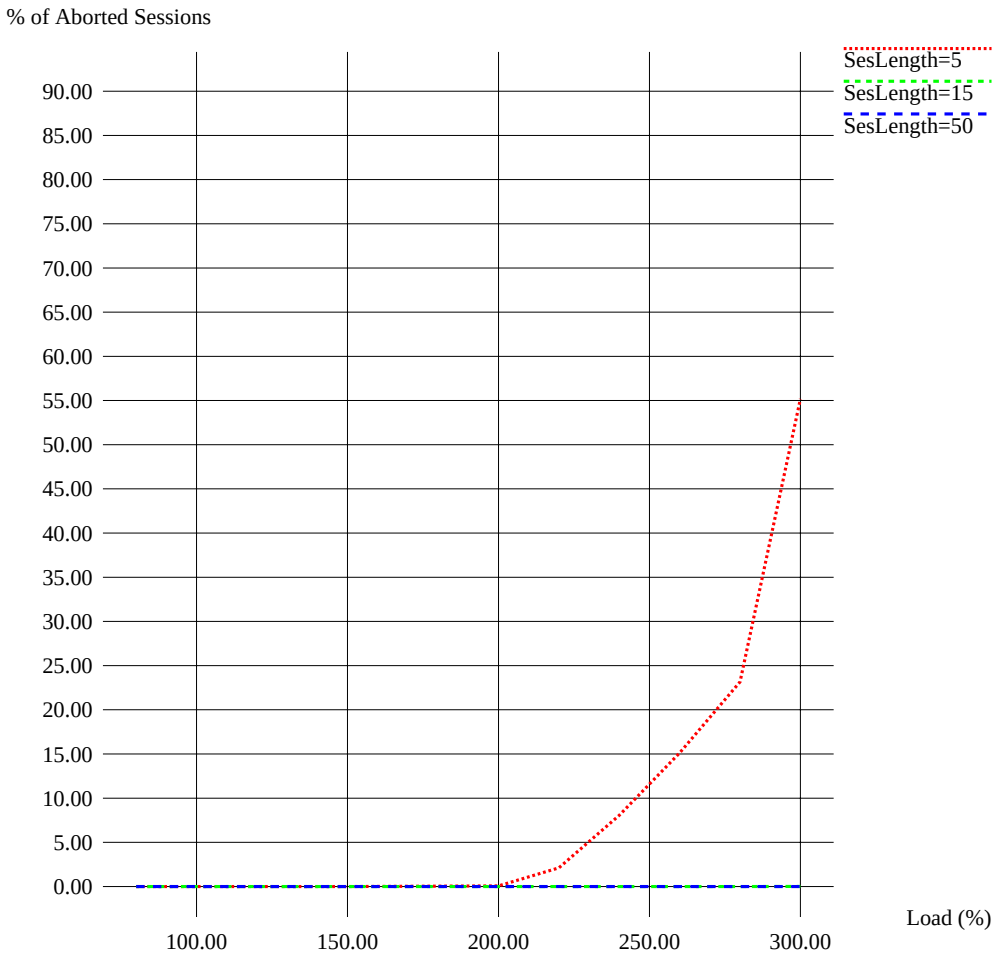

Figure 5: Percentage of Aborted Sessions from Admitted for Processing by Server with Admission Control.

centage of aborted sessions, i.e. those which server has admitted for processing and failed to complete. The percentage of aborted sessions characterize a level of service server is able to provide. The results for sessions with a mean of 15 and 50 are perfect across the whole load space. They meet the desired level of service requirement: zero aborted sessions from those accepted for service.

For a workload with mean of 5, the results are getting worse at load greater than 200%. At 300%, up to 55% of admitted for processing sessions are aborted. And it is happening even when we are using ac-responsive strategy, which provides us with most restrictive admission policy. The reason is that the shorter the average session length – the higher the number of sessions generated by the clients and accepted by the server during the ac-interval (i.e. 1 second). For example, if a web server is in "accept mode" then for a load of 300%, during one second it accepts around 600 new sessions, in addition to the sessions which are already in progress.

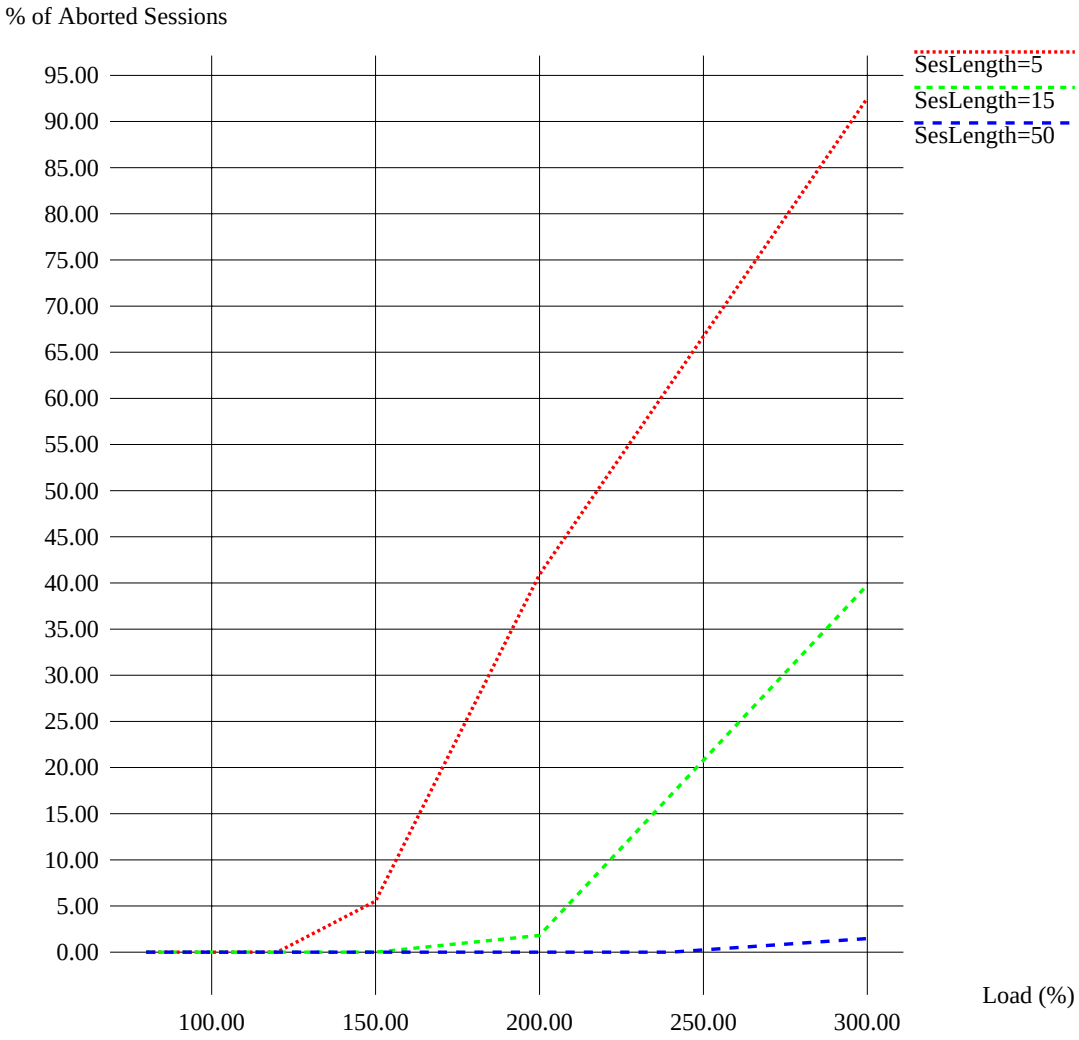

Figure 6: Percentage of Aborted Sessions from Admitted for Processing for Server with Admission Control, AC-Interval = 5sec, Workload with Average Session Length of 5.

The main reason for aborted sessions under this scenario is that the listen queue overflows. One way to fix the problem is to reduce the ac-interval. The percentage of aborted sessions, for a server with an admission control mechanism using an ac-interval of 0.5 seconds, drops to 7.5%. It is much better but still might be not acceptable for some of the applications. Moreover, it is not always possible to reduce ac-interval to desirable value. For some operating systems, cpu utilization is updated on a base of 5 seconds interval. However, as it is shown in [CP98], the results for SBAC based on ac-interval of 5 seconds are unsatisfactory for a whole family of workloads with an average session length less than 50. Figure 6 shows the percentage of aborted sessions from those admitted for processing for an admission control mechanism with an ac-interval of 5 seconds.

A more general reason why a CPU utilization based implementation of *SBAC* can break under

certain rates and not work properly, is the following. The decision, whether to admit or reject new sessions, is made at the boundaries of ac-intervals. And this decision can not be changed until the next ac-interval. However, in presence of a very high load, the number of accepted new sessions may be much greater than a server capacity, and it inevitably leads to aborted sessions and poor session completion characteristics.

The only way to avoid the situation, described above, is:

- to estimate the number of sessions a server is able to process, and
- to admit during the time interval no more sessions, than the estimated number prescribes.

7 Predictive Admission Control Strategy

In order to correctly estimate the number of sessions a server is able to process per time interval, we also need to evaluate the session rejection overhead. For workloads with short and medium average session length, the rejection overhead can be significant for high traffic loads. But even when it is 5% to 10% only, this overhead should be taken into account, since a small inaccuracy tends to accumulate over longer period of time.

This section derives a worst case bound to estimate the rejection overhead as a function of the applied load and average session length.

We use the following denotations:

- S_r - a server capacity in requests, i.e. number of connections (requests) per second a server can sustain.
- S_s - a server capacity in sessions, i.e. number of sessions per second a server can complete.
- $SesLength$ - an average session length.
- $Load$ - an applied load in sessions ($Load = 2$ means a load of 200% of server capacity).
- x - a number of rejected sessions per second.
- y - a number of completed sessions per second.

First of all, there is a simple relation between S_r, S_s and $SesLength$:

$$S_s = \frac{S_r}{SesLength} \qquad (1)$$

Since S_s is a server capacity in sessions and $Load$ is an applied load in sessions, $Load * S_s$ is a total number of issued sessions per second. Obviously, the sum of completed and rejected sessions per second is a number of sessions in total, a server has received per second:

$$x + y = Load * S_s \qquad (2)$$

There are two types of sessions: completed and rejected ones. Each completed session implies that a client consequently makes, on average, the number of requests defined by the $SesLength$. Each rejected session is equivalent to processing a single request - a worst case estimate of the cost of sending an explicit rejection message to the client. Thus a number of requests per second handled by a server is defined in the following way:

$$y * SesLength + x = S_r \qquad (3)$$

Replacing S_s in (2) with a formula (1), and expressing y from (2), we have the following equation:

$$y = \frac{Load * S_r}{SesLength} - x \qquad (4)$$

Replacing y with (4) in equation (3) we can express x:

$$x = \frac{S_r * (Load - 1)}{SesLength - 1} \qquad (5)$$

Since x is a number of rejected sessions (rejection messages) per second, and S_r defines a total number of requests per second processed by a server, then a percentage of rejection messages from the total number of requests is defined as follows:

$$\frac{100\% * x}{S_r}$$

Let us call this percentage the $RejectionPercentage$. Here is the final equation:

$$RejectionPercentage = 100\% * \frac{(Load - 1)}{SesLength - 1} \qquad (6)$$

The rejection overhead depends on average session length and the load received by the server. Figure 7 illustrates the rejection overhead as a percentage of rejection messages to a total number of requests per second.

The rejection cost varies depending on the average session length and applied load: the higher the load and the shorter the session length – the higher the rejection overhead. However, for most of the load values and workloads of interest – the overhead is less than 10%.

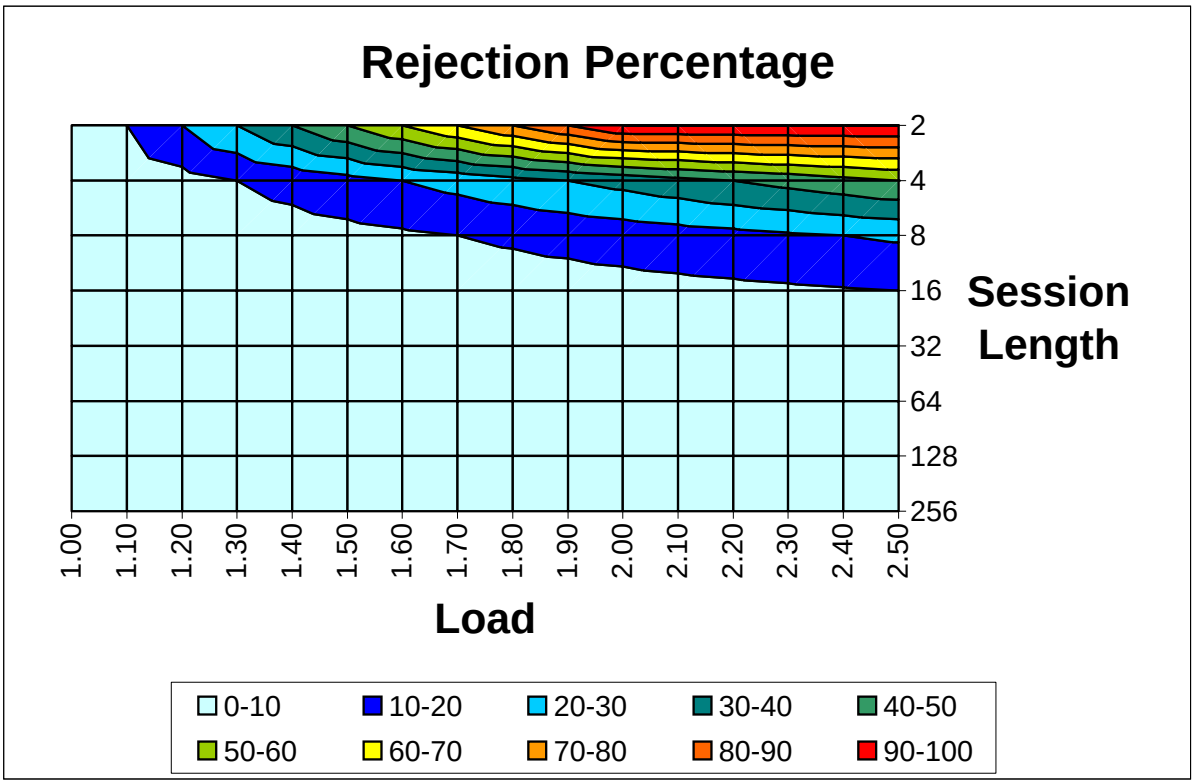

Figure 7: Rejection Cost as a Percentage of Rejection Messages to a Total Number of Requests per Second.

REMARK: Formula (6) holds for the *Load* and *SesLength* values, satisfying the following condition: $Load - 1 \leq SesLength - 1$. For the other values, formula (6) is meaningless and reflects the situation that the applied load is so high that the server's capacity is not enough to send all the rejection messages.

For example, let us consider a server with a capacity of 1000 requests per second: $S_r = 1000$, and let an average session length be 5: $SesLength = 5$. Then a server capacity in sessions is 200: $S_s = 200$ accordingly to (1). Load of 500% will produce 1000 sessions per second which is a maximum request rate server can sustain. Thus all the server capacity will be consumed sending the rejection messages. The same value is produced by the formula (6) computing 100% of rejection cost.

Once we have estimated rejection overhead, it is easy to predict the number of sessions a server is capable to process per time interval. It is derived by replacing x with (5) in equation (4):

$$y = \frac{S_r * (SesLength - Load)}{SesLength \times (SesLength - 1)} \qquad (7)$$

Predictive admission control strategy works in the following way. For each ac-interval T_i it predicts the number of sessions a server is able to process, and the web server accepts this quota, and reject any new session above those quota.

Formula (7) depends on three parameters: S_r -request rate per second a server can process, *Load* - new sessions arrival rate, and *SesLength* - an average session length.

How these parameters can be obtained in practice?

Request rate per second S_r a server can process (for this particular workload) is easily measured parameter.

A running counter of accepted sessions C_s (C_s increments for each accepted session by one) and a running counter of requests C_r related to the accepted sessions (C_r increments for each processed request belonging to an accepted session), allows an approximation of the average session length to be computed $SesLength = \frac{C_r}{C_s}$.

Once, the average session length is evaluated, the S_s - a server capacity in sessions, can be computed using (1).

And after that, by counting the number of new sessions arrivals, *Load* can be evaluated.

An approximation of *ac-cycle*, discussed in Section 5, can be done by measuring inter request time (it is, essentially, a sum of the request response time and a client think time) multiplied by the average session length.

REMARK. Clearly, the efficiency of predictive strategy depends on an accuracy of our prediction. The strategy works much better when one keeps track of possible inaccuracy occurred, for example, as a result of rounding up fractions. More serious source of inaccuracy can occur because of mispredicting the *Load*, since our prediction is based on the previous interval. For example, the *Load* during the previous ac-interval was 200%, and we estimated using (7) how many sessions can be accepted during the following time interval. However, later analysis of the *Load* during this time interval, shows that it was 300%. It leads to some mismatch, easily computed using formula (7): we accepted slightly more sessions than is allowed, since we assumed slightly smaller rejection overhead (or situation can be vice versa). In order to eliminate further accumulation of such inaccuracy, next ac-interval quota has to be adjusted (increased or decreased) by the computed sessions amount.

In our simulation model, we implemented predictive strategy which adjust possible inaccuracy as well as evaluates an amount of unused quota for the last few ac-intervals to allow its usage in near future.

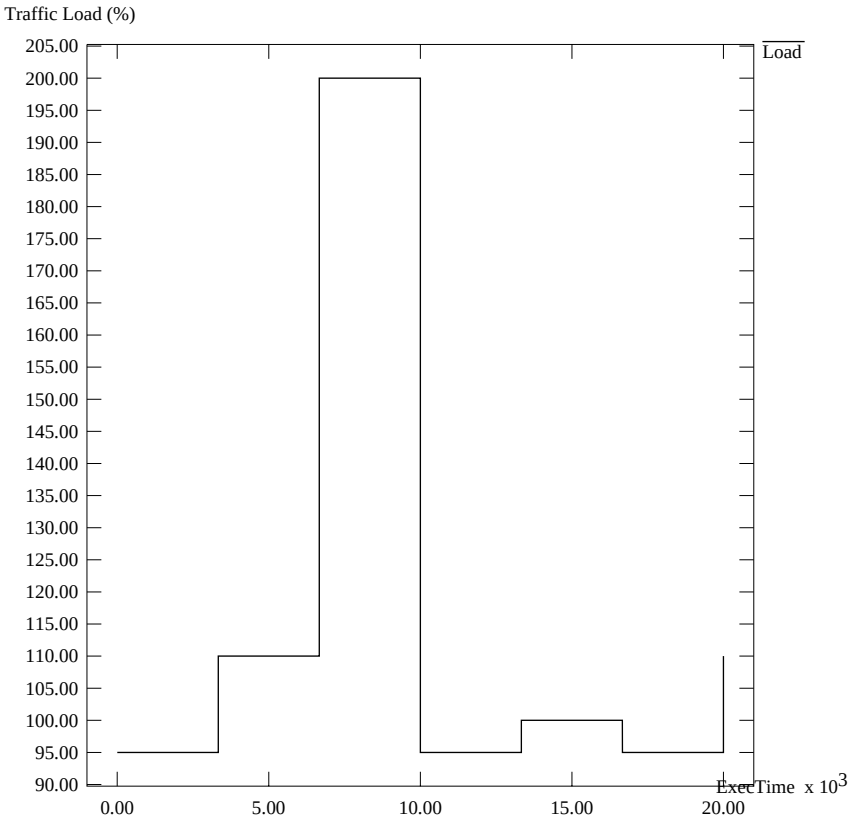

Figure 8: "Usual Day" Workload Traffic Pattern

8 Comparison of Admission Control Strategies

In this section, we analyze and compare different admission control strategies: ac-stable, ac-responsive, hybrid and predictive.

New strategies: hybrid and predictive, are designed to complement shortages of ac-stable and ac-responsive strategies. Since these shortages show up under different load conditions, we designed two variable traffic patterns to verify whether new admission strategies adequately adjust their behaviour depending on traffic rates.

First traffic pattern is defined by the pattern showed in Figure 8. We call it the *"usual day"* traffic pattern. It has only a few intervals of not very high overload, and the rest of the time, it is a load close to the server capacity. This type of load might be typical in practice: most of the time, the load is manageable, only occasiongly exceeding server capacity.

Second workload is defined by the pattern shown in Figure 9. We call it the *"busy day"* workload. This traffic pattern spends a half of the time in overload (reaching a peak of 300% during one of the intervals), and for the other half of the time it has a load close to a server

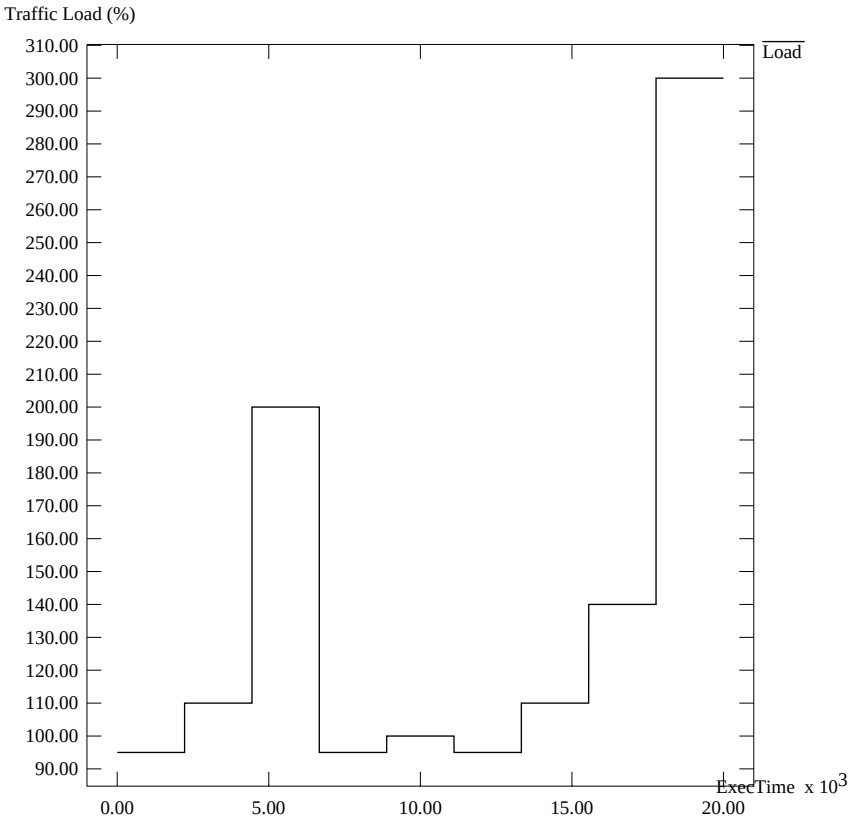

Figure 9: "Busy Day" Workload Traffic Pattern

capacity.

We do not include a "bad day" workload (with consistently high overload for all intervals) since the results are predictable, and we will comment them at the end of the section.

Figures 10, 11, 12 show the results for a *"usual day"* traffic pattern: both server throughput in completed sessions and percentage of aborted sessions for workloads with average session length of 5, 15, and 50 correspondingly.

As it was expected, for a workload with average session length of 5, even for "usual day" traffic load, ac-stable strategy has 13.5% of aborted sessions (from accepted ones), while ac-responsive strategy has no aborted sessions, but its throughput is 6% less than throughput of ac-stable strategy. Hybrid strategy has the same as ac-stable strategy throughput (even slightly better) but only 1.5% of aborted sessions. Thus, the proposed hybrid strategy improves server throughput while supporting high levels of service: very low number of aborted sessions. The predictive strategy outperforms all of the strategies: it improves server throughput by 14% comparing with ac-responsive strategy and has no aborted sessions.

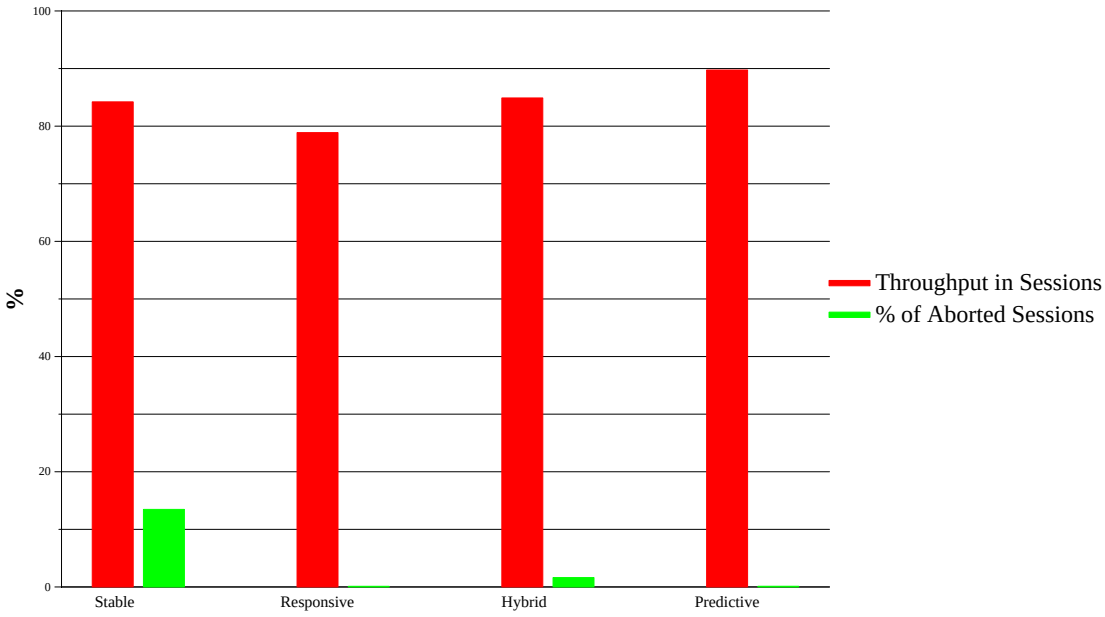

Figure 10: "Usual Day" Workload with Average Session Length of 5.

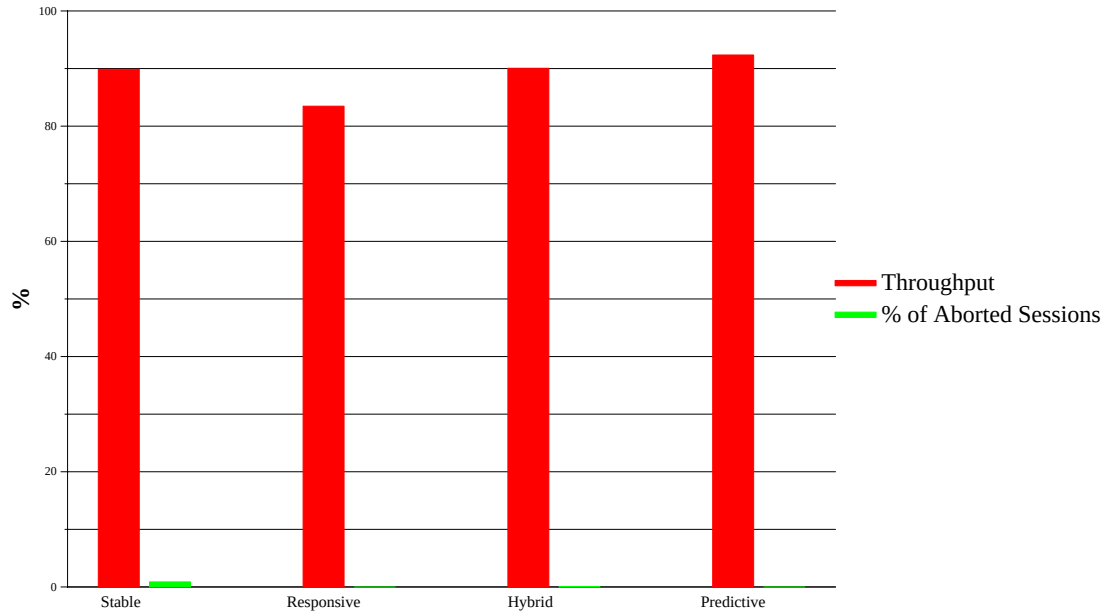

Figure 11: "Usual Day" Workload with Average Session Length of 15.

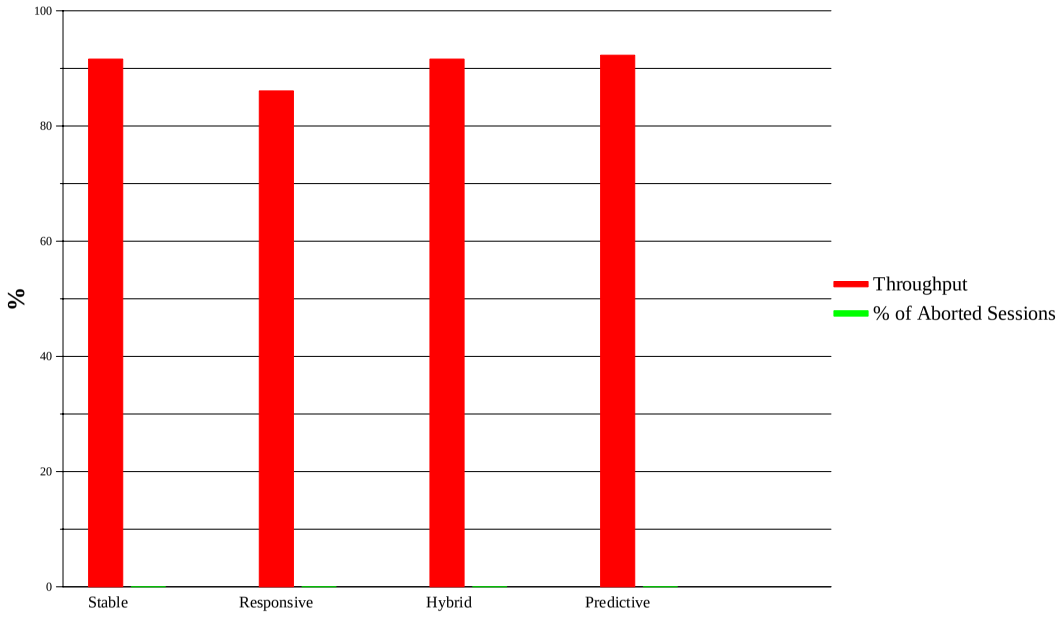

Figure 12: "Usual Day" Workload with Average Session Length of 50.

Simulation results for workloads with average session length of 15 and 50 are similar. The rates of aborted sessions are significantly less for all the strategies. All of the strategies are able to provide high levels of service. However, the hybrid and predictive strategies support higher server throughput in completed sessions.

Figures 13, 15, 16 show the results for a *"busy day"* traffic pattern and workloads with average session length of 5, 15, and 50 correspondingly.

For a *"busy day"* traffic pattern the number of aborted sessions is higher, especially for a workload with average session length of 5. For this workload, ac-stable strategy has 27% of aborted sessions. Even ac-responsive strategy is unable to provide satisfactory level of service: it has 13% of aborted sessions. Since hybrid strategy is a special combination of ac-stable and ac-responsive strategies, it also has 13% of aborted sessions, with, however, higher sessions throughput. Thus, none of these three strategies provides an acceptable level of service for workloads with short average session length.

The predictive strategy produces the best results. It provides both: the best overall server throughput (14% improvement) while having no aborted sessions (or almost no aborted sessions: 0.2%).

Simulation results for workloads with an average session length of 15 and 50 are similar. The rates of aborted sessions are less for all the strategies. Only ac-stable strategy fails to provide adequate level of service: it still has 13% of aborted sessions. However, hybrid strategy improves the situation: it has only 1.6% of aborted sessions and a 6% improvement

in throughput.

The predictive strategy, again, provides the best overall results.

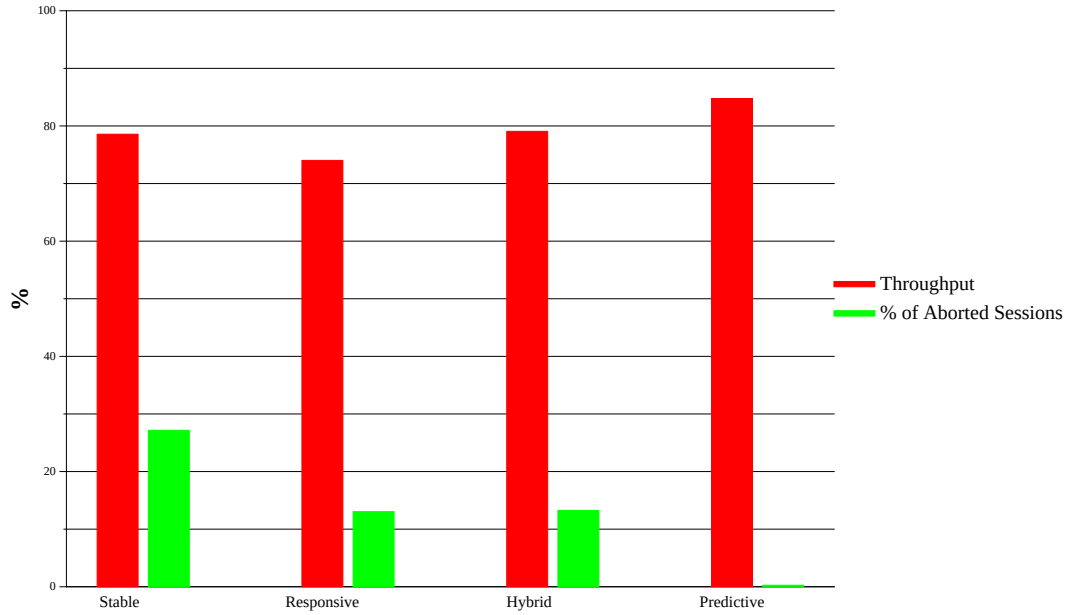

Figure 13: "Busy Day" Workload with Average Session Length of 5.

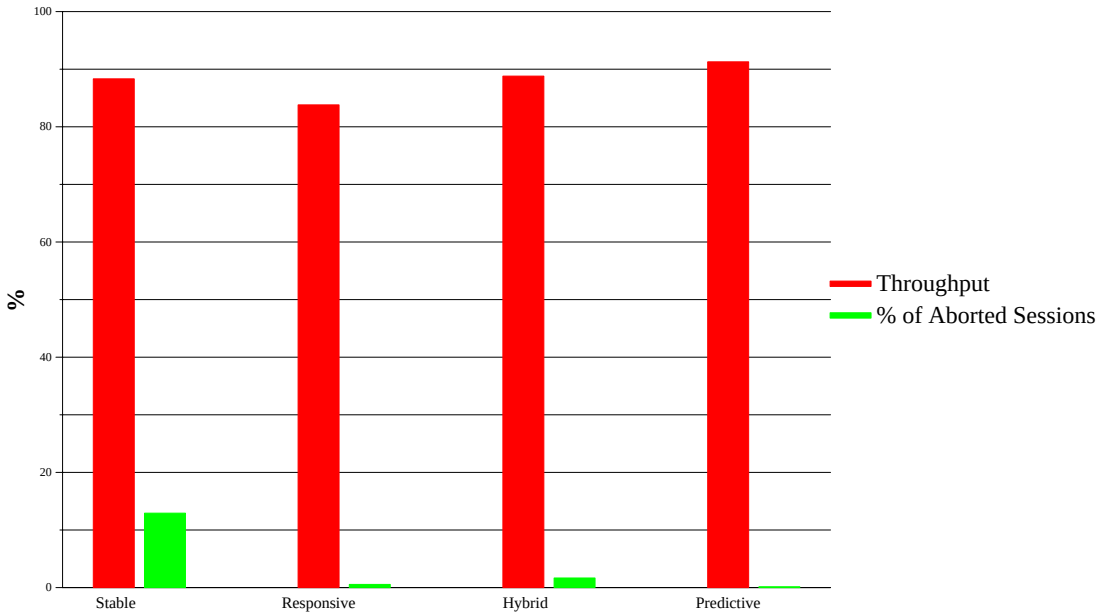

Figure 14: "Busy Day" Workload with Average Session Length of 15.

Figure 15: Rejection Cost as a Percentage of Rejection Messages to a Total Number of Requests per Second.

For a "bad day" traffic pattern (with consistently high overload for all intervals) the results

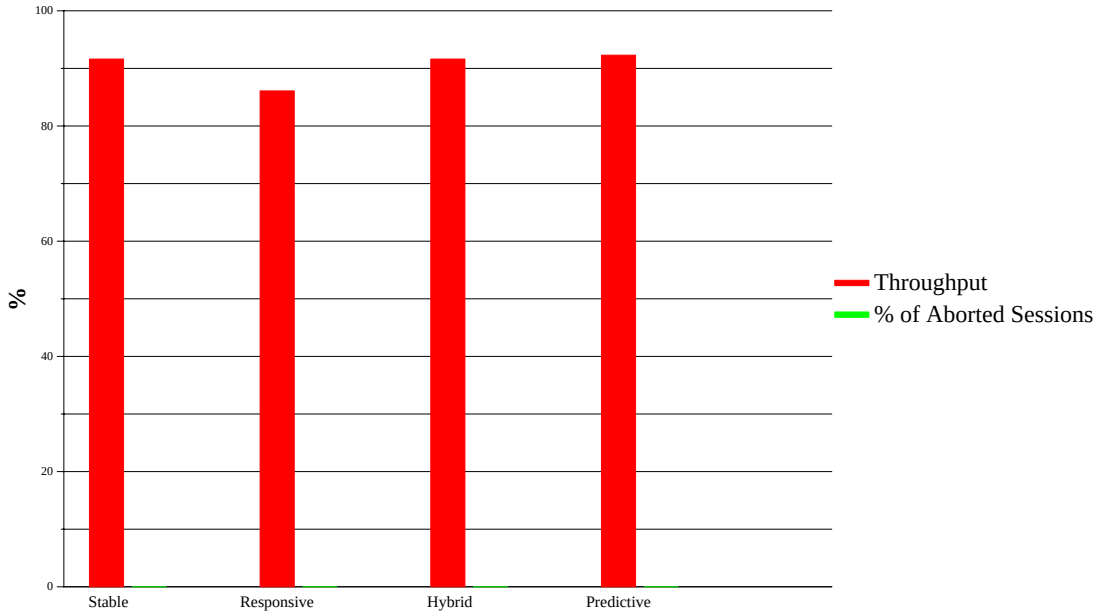

Figure 16: "Busy Day" Workload with Average Session Length of 50.

are predictable from [CP98] and show the same tendency, observed for a "busy day" traffic pattern. For workloads with short average session length, the only strategy, which works consistently well, is predictive one. For workloads with medium and long average session length, both hybrid and predictive provide the best results.

9 Conclusion

In this paper, we propose two new admission control strategies: hybrid and predictive ones, aiming to optimize the performance of *SBAC* mechanism and improve the level of service (number of aborted sessions) provided by SBAC. The hybrid strategy is based on a self-tunable admission control function adjusting itself according to variations in traffic loads. We show that the proposed hybrid strategy successfully combines the most attractive features of both ac-responsive and ac-stable policies. Compared to ac-responsive and ac-stable strategies, it shows improved throughput (up to 10%) and high level of service (less than 1% of aborted sessions) for workloads with medium to long average session length.

The predictive strategy evaluates the number of new sessions that can be accepted while still guaranteing that all future session requests will be processed. It consistently shows the best performance result for different workloads and traffic patterns. It improves server throughput (up to 15%) while maintaining a high level of service (less than 0.2% of aborted sessions).

The proposed hybrid and predictive admission policies allow the design of a powerful admission control mechanism which tunes and adjusts itself for better performance across different workload types and different traffic loads.

10 References

[CP98] Cherkasova, L., Phaal, P. Session Based Admission Control: a Mechanism for Improving the Performance of an Overloaded Web Server. HP Laboratories Report No. HPL-98-119, June, 1998. URL: http://www.hpl.hp.com/techreports/98/HPL-98-119.html

[Schwetman95] Schwetman, H. Object-oriented simulation modeling with C++/CSIM. In Proceedings of 1995 Winter Simulation Conference, Washington, D.C., pp.529-533, 1995.

[SpecWeb96] The Workload for the SPECweb96 Benchmark. URL http://www.specbench.org/osg/web96/workload.html

[WebStone] WebStone: The Standard Web Server Benchmark. URL http://www.mindcraft.com/benchmarks/webstone/