Detecting Timed-Out Client Requests for Avoiding Livelock and Improving Web Server Performance

Richard Carter and Ludmila Cherkasova
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303
e-mail: {carter,cherkasova}@hpl.hp.com

Abstract.

A Web server's listen queue capacity is often set to a large value to accommodate bursts of traffic and to accept as many client requests as possible. We show that, under certain overload conditions, this results in a significant loss of server performance due to the processing of so-called "dead requests": timed-out client requests whose associated connection has been closed from the client side. In some pathological cases, these overload conditions can lead to server livelock, where the server is busily processing only dead requests and is not doing any useful work. We propose a method of detecting these dead requests and avoiding unnecessary overhead related to their processing. This provides a predictable and controllable platform for web applications, thus improving their overall performance.

DTOC strategy is implemented as a part of WebQoS for HP9000 [HP-WebQoS].

Contents

1 Introduction

The World Wide Web is experiencing a phenomenal growth. Thousands of companies are deploying web servers, with some web sites becoming extremely popular. To function efficiently, these sites need fast, high-performance HTTP servers. The listen queue capacity of these servers is often set to a large value to accommodate bursts of traffic and to accept as many client requests as possible. In servicing these requests, web applications often provide both static and dynamic content, performing complex database queries and other data-manipulation. This can lead to a large variance in request service times. Congested and overloaded Internet routes only add to this variance: the download time for a given document can range from 5% to 500% of its typical latency [CI97].

Long delays typically cause human clients to cancel and possibly resubmit their requests. For a server with a small queue capacity, this client request timeout probably occurs during the time when the client is competing to get the request into the server's request queue. If the timeout occurs before the client request can enter the queue, there is no residual effect in the server. However, with larger queue capacities and with sufficient request loads, it is increasingly more likely that the request timeout will occur with the request sitting in the server's request queue. In many systems, the client cannot respond to this timeout by removing the request from the server's queue and either cannot or does not choose to notify the server that the request's response is no longer needed. When a server processes these timed-out requests, it is doing no useful work and is, instead, wasting its critical resources. One could picture a scenario, with a very large and full request queue, in which all client requests timeout before being processed by the server. In this case, all the server's resources are used to process timed-out requests and no server resources are applied to "still-vital" requests. We term this pathological system state *request-timeout livelock*.

In this paper, we propose a method to detect timed-out client requests, and use it to improve server performance. Detection of timed-out client requests (DTOC) imposes little additional overhead. However, to minimize this overhead further, we propose to perform this detection only in the presence of overload. Using simulation models, we show how DTOC can improve server performance and avoid livelock. We also use analytical methods to corroborate some of these simulation results.

We consider the DTOC technique as complementary to session based admission control (SBAC), introduced in [CP98]. Additionally, DTOC can improve SBAC performance for workloads with short average session length, where for high traffic loads there is a large percentage of retried clients requests, as was shown in [CP98].

DTOC strategy is implemented as a part of WebQoS for HP9000 **??**.

2 Methods for Detecting Timed-Out Client Requests

If a client experiences a long response delay after sending a request to a web server, there are three possible behaviors the client could exhibit:

- "patient behavior": the client waits patiently for the response no matter how long it takes;
- "anxious behavior": the client is anxious to get the response and clicks the browser "stop" button, followed by the browser "reload" button to resend the request;
- "impatient behavior": the client is not tolerant of the delay, clicks the browser "stop" button and leaves the site, loosing interest in the content because it took too long to receive.

Since timed-out requests are not removed from the listen queues of current web servers, their processing could lead to a substantial waste of server resources. Based on this, the first client behavior, the "patient behavior", is the most desirable from the standpoint of preserving web server efficiency. Unfortunately, this is not the most typical client behavior. More often, clients exhibit the "anxious" or "impatient" behaviors. Under these last two scenarios, an overloaded web server could end up processing a lot of "dead", timed-out requests. While the web server is processing these dead requests, wasting its resources on useless work, the "still-vital" requests at the end of the listen queue encounter ever-longer delays that exceed their client's patience threshold. This creates a snowball effect in which all requests timeout before being serviced. All the server's resources are used to process timed-out requests, thus resulting in *server livelock*, where the server is "busily" processing only dead requests and is not doing any useful work. We term this form of server livelock *request-timeout livelock.*

In practice, request-timeout livelock is not an easily recognizable situation. Typically, servers are able to detect that the client is no longer awaiting the response because the action of sending the response reveals a closed client-server connection. This realization saves the server from the task of sending the entire response back to the client. The obvious disadvantage of this solution to server livelock is that the server work involved in preparing the response is not avoided. Server resources are still wasted on preparing useless responses and request-timeout livelock scenarios still exist.

At a high level, server methods for detecting timed-out client requests must rely on one of two approaches:

1. the server can initiate an explicit communication back to the client, asking the client if it still desires a response to the request, or

2. the server can infer a client's interest in the response from the health of the client-server connection. For web applications, if the client has closed the connection, it can be assumed that the client no longer desires a response to the request that was conveyed over that connection.

Web servers communicate with clients via HTTP [HTTP1.1] over TCP/IP-based networks. We present several methods for how web servers can check for timed-out client requests in this environment.

The TCP/IP client-server connection, or socket, can be thought of as consisting of two unidirectional channels. For some protocols, such as "remsh", the client can do a "half-close" of the client-to-server channel even though the client still desires and expects a response back from the server on the still-open server-to-client channel. Luckily, the HTTP1.1 spec does not discuss the possibility of this "half-close": a client should keep both channels open if it still desires a response and should close both channels otherwise. With TCP/IP, the client's client-to-server channel close is communicated via a FIN packet to the server, which results in the connection transitioning to the "CLOSE_WAIT" state. Thus, the server application can tell that the client is no longer waiting for the response by detecting this CLOSE_WAIT state change within the server OS's network software. Here are three server-side approaches toward that end:

1. **"Peek" at the socket read buffer state**
   This can be accomplished on a computer running HP-UX 10.20, at a point when the client request had been read, by performing the following software steps:

   (a) place the socket connection in NON-BLOCKING mode using `fcntl()` or `ioctl()`;

   (b) do a `recv()` call using a `flags` parameter value of `MSG_PEEK`;

   (c) restore the socket to its prior mode.

   The `recv()` call produces a return code that indicates the number of bytes read, which would be 0 for the case of a closed connection. In contrast, the return code for an open connection with no bytes waiting is -1 with `errno == EAGAIN`. If the client had sent additional bytes past the logical end of the request, as might be the case for pipelined requests on persistent connections, the return code would be greater than 0. Note that the `MSG_PEEK` setting of the `flags` parameter avoids disturbing these bytes in the input buffer. In summary then, the return value of the `recv()` call can be used to uniquely distinguish the closed client-to-server connection case (and hence a timed-out request).

2. **Explicitely request the socket state**
   The `getsockopt()` system call is commonly used by applications to probe a socket's status and control settings. Unfortunately, the standard functionality of this call does not

provide access to the TCP/IP “state-diagram” state. Luckily though, the HP-UX 10.20 operating system has extended `getsockopt()` in OS patch PHNE_14472 to provide this functionality. With this patch, the following call returns the socket state in the variable tcp_state:

```
getsockopt(sockfd, IPPROTO_TCP, TCP_STATE, (char *) &tcp_state,
                    &tcp_state_len)
```

The test `(tcp_state==TCPS_CLOSE_WAIT)` can be used to detect the closed client-to-server connection. Note that this `getsockopt()` extension may not be available in HP-UX 11.0.

3. **Register a call-back function for a socket's transition to `CLOSE_WAIT`**
This method would require customizations that are not currently part of the HP-UX 10.20 operating system. One option would be to build on the Unix signal-handling mechanisms by defining a new signal that would be generated when a socket transitions to the `CLOSE_WAIT` state. Another option would be to augment the `setsockopt()` function to set up a call-back function as part of the socket descriptor state. In both cases, the call-back function must know either implicitly or through passed parameters which socket and process/thread are involved. This method could, in theory, have performance advantages over the previous methods, since no unnecessary connection state polling is done. However, the overhead to set up and remove the call-back function may be comparable. This method may also provide unique performance advantages in the presence of client requests that timeout during their response preparation. More will be said on this after we present our simulation results.

Although we have presented three methods for detecting closed client-server connections, there could indeed be other approaches. Different operating systems and implementations of the transport layer and below will offer different opportunities for detecting closed client-server connections. In addition to this issue of *how* to check for a client request timeout, there are the issues of *when* and *how often*. For example, a server could potentially check for a client request timeout at one or more of the following times:

- when the request is first pulled off of the request queue, and/or
- when the request has been parsed (and the amount of server work to generate the response can be estimated), and/or
- periodically throughout the preparation of the response.

As a final enhancement, maintaining statistics of when clients timeout for various types of requests could be useful in forming a more-optimal strategy for checking timed-out requests.

To summarize, we have shown a number of ways that a web server can check for closed client-server connections in order to detect timed-out client requests. The important advantage of this approach is that it permits servers to detect that the response to a client's request is no longer needed before much of the server resources to generate that response have been expended. This results in an increase in server efficiency. In addition, no prior work has addressed the pathological system behavior we term request-timeout livelock, in which all of the server's efforts are spent preparing unneeded responses. Under the usually-valid assumption that the request-timeout check is far easier for the server to accomplish than preparing the response, this method provides protection against the request-timeout livelock situations.

3 Simulation Model

In order to demonstrate the performance impact of processing undetected timed-out requests in an overloaded web server as well as to demonstrate how it can lead to a request-timeout livelock, we built a simulation model. Sections 3.1 and 3.2 that follow discuss this in more detail.

3.1 Workload Model

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

For example, the SpecWeb96 file mix is defined by a response-file distribution based on 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 connections/sec supported by a server when retrieving files of the required file mix.

The primary limitations of today's web benchmarks are considered to be the following [MCS97]:

- they do not accurately reflect real usage patterns;
- they do not include requests that generate errors or involve dynamic responses;

- they focus on throughput measurements to the exclusion of latency considerations.

Fortunately, to demonstrate the problem with timed-out requests, we do not need a response-file distribution that perfectly reflects today's usage patterns. It is merely sufficient that the workload have a reasonably high variance of response file size, as do today's usage patterns. However, it is also important to realize that there is a high variance in the server CPU time needed to prepare responses. This observation exposes an additional flaw in today's benchmarks. They simply do not capture the wide variety of requests and responses that servers observe and generate. For example, most benchmarks do not include dynamic CGI-created responses, which typically are much more resource-intensive (CPU-consuming) for a server to perform. This translates directly into a server with increased request service times and a lower supported throughput in requests/sec. Often a web server with a specification of 1000 requests/sec for SpecWeb96 will sustain 100 or less requests/sec of dynamic content. That brings us to a discussion of user-perceived response time. From analysis of web server access logs, 80% of users who experience latencies of greater than 15 seconds immediately leave the site [M97]. This underlines the importance of modeling end-to-end latency and the various client behaviors associated with high latencies.

In our simulation model, we decided to use a SpecWeb96-like file distribution for the workload because it has a high variance of requested file sizes. Since the service time in our model is proportional to the requested file size, we can use this distribution to demonstrate the effect of timed-out requests. One can reason that processing large and medium-sized files is equivalent to running CGI scripts requiring the same amount of time (since in the model, the requests only differ by the service time and the client that issued the request). Further details of the workload are available in Appendix A.2.

In our simulation, a request is defined as a structure with the following parameters:

- the client which originated the request;
- the requested file size;
- a time stamp of when the request was issued.

In order to reflect the server running a mix of dynamic and static content, we set the server capacity to 100 connections/sec.

3.2 Server Model: Functionality and Basic Parameters

We built a simulation model using C++Sim [Schwetman95] to better understand the different web server behaviors that exist under request-based and session-based workloads. The basic structure of the model is outlined in Figure 1.

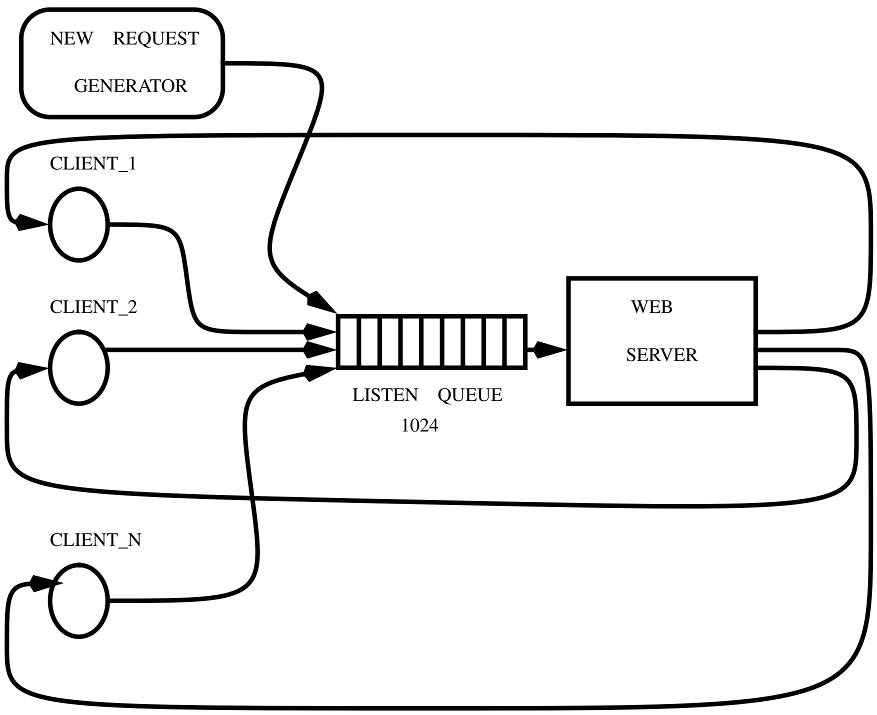

Figure 1: Basic Structure of the Simulation Model.

It consists of the following components:

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

The workload generator produces a stream of new requests according to specified load parameters. We are able to use an open model for this request generation. Each generated request is sent to the web server and is stored in the server's *listen queue*. We limit the size of the listen queue to 1024 entries which is a typical default value. Any subsequent request-retries are issued and handled by an individual client. The client behavior is defined by a closed-loop model: the client issues the next request-retry only when it does not receive a response from the previous request in a predefined timeout period.

At the model output, we partition all requests into two groups:

- *successfully-completed requests* – requests for which responses were received by the clients in time (but possibly after one or more retries);
- *unsuccessful requests* – requests for which responses were not received in time (even after issuing a predefined number of retries) or requests that encountered a full listen queue.

There are two reasons for marking a request *unsuccessful:*

- the server's listen queue was full and the client's connection attempt was refused by the server;
- the server was not able to produce a response before the request or any of its retries timed-out. More specifically, after issuing the request, the client will first wait for a server response for a predefined timeout period. If this period elapses with no response, the client times out and resends the request. If after a limited number of these *retries*, the response still has not been received, the request is considered to be unsuccessful.

Note that a client's first task in sending a request is to establish a connection with the server. If the client's first connection attempt is ignored by the server due to a full listen queue, the client will typically try this connection attempt again after some delay. After some number of unsuccessful connection retries, a "connection refused" message is often then presented to the user, who may then restart the whole procedure. Note that we decided to simplify our simulation model by pronouncing the request *unsuccessful* upon the first failed connection attempt, without additional client retries.

Traditionally, web server performance is measured as the throughput of processed requests. We propose however to measure web server performance as the throughput of only the *successfully-completed requests*. This definition allows us to highlight the difference between the total work performed by a server and its useful part. It also allows us to clearly identify server livelock situations where the observed server utilization is close to 100% but with no successfully-completed requests (i.e. with only unsuccessful requests) processed.

For our simulations, we modeled two different server strategies:

- *a regular strategy*, in which the server processes requests without detecting whether they are timed-out or not;
- *a DTOC strategy*, in which the server first checks whether a client request has timed-out or not and further processes only those requests that have not timed out.

The detection of timed-out client requests (DTOC) imposes little additional overhead. In our simulation model, and based on measurements of real machines, we set the overhead to be 0.5% of the service time to process an average request. In order to further minimize this overhead, we propose to perform the checking of requests only during periods of overload. Toward that end, we measure the server utilization each second. If the server utilization is above 95% then we start using the DTOC strategy and begin checking whether the client requests are timed-out or not. Once the server utilization drops below 95%, we revert the server back to its regular strategy.

4 Simulation Results

4.1 Using the DTOC Strategy for Avoiding Livelock and Improving Server Performance: Throughput as a Function of Traffic Load

In this section, we compare the performance of a web server augmented with the DTOC strategy against a web server using the regular strategy. We vary some of the parameters of interest to see their impact on the simulation results. Other parameters remain constant across all simulation runs: the workload is the SpecWeb96-like filesize distribution described in Section 3.1 and Appendix A.2 and the server capacity is fixed at 100 requests/sec while processing this mix. The server's listen queue size is fixed throughout at 1024 entries.

Figure 2 shows the performance of a server using the regular strategy. The horizontal axis shows the applied load, represented as a percentage of the maximum server capacity: an applied load of 300 represents 300% of the maximum load that the server can process. Since our server capacity is 100 requests/sec, a load of 300 coincidentally implies a load of 300 requests/sec. The vertical axis of Figure 2 shows the useful server throughput, measured as the rate of processing successfully-completed requests, again normalized to the server capacity. A throughput rating of 50 implies a server that is processing 50% of its maximum load, which is coincidentally 50 successfully-completed requests/sec.

Also in Figure 2, we show the simulation results for four different values of client request timeout, namely, 5, 8, 9, and 10 seconds. In addition, we use clients that perform no retries, i.e. if the response is not received within the predetermined timeout period, the client does not resend the request.

The results show that the number of successfully-completed requests under loads higher than 100% drop significantly. For clients with a request timeout of 5 seconds, the server quickly enters a request-timeout livelock state and cannot recover. In this state, in spite of the server utilization being 100%, the *useful* server throughput is zero. With the client timeout set to 8 seconds, the server performance is slightly improved. After entering request-timeout livelock, the server is able to recover, but only for a time: the server ends up oscillating back and forth between periods of livelock and non-livelock.

Finally, for clients with a timeout set to 10 seconds, the useful server throughput under high loads is around 40%. This result is plausible given the following reasoning: Under high loads, the listen queue will almost always be fully populated with requests. A newly arriving request that just fills the 1024-entry listen queue must wait 1025 service times before its response is complete. At a service rate of 100 requests/sec, this takes an average of 10.25 seconds. Since 50% of such requests will experience a wait of less-than 10.25 seconds, it is plausible that 40% might experience a wait of less-than 10.0 seconds (the critical timeout value).

Figure 3 shows the performance of a web server augmented with the DTOC strategy and with

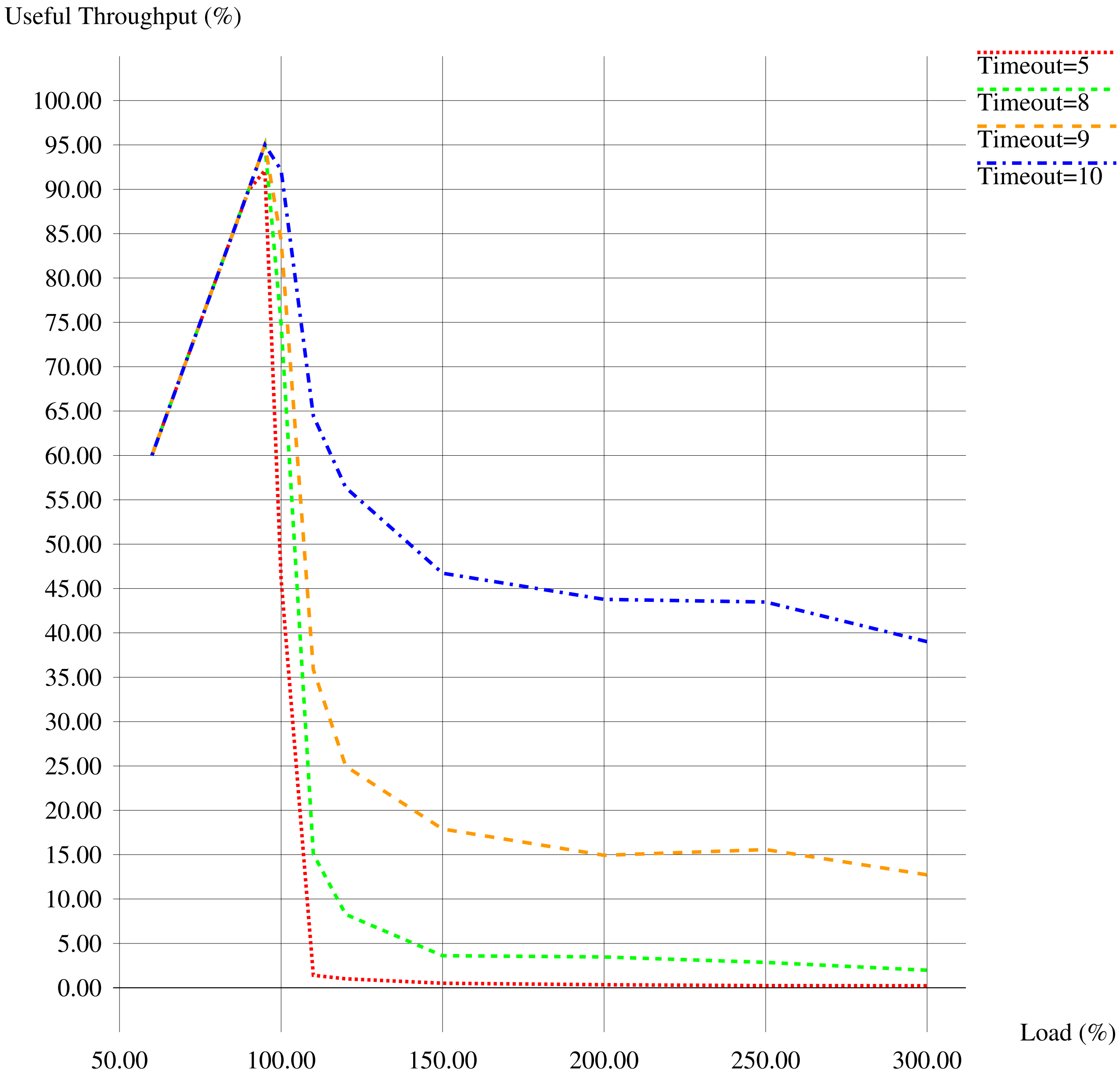

Figure 2: Throughput for a Server using the Regular Strategy with Clients with No Retry.

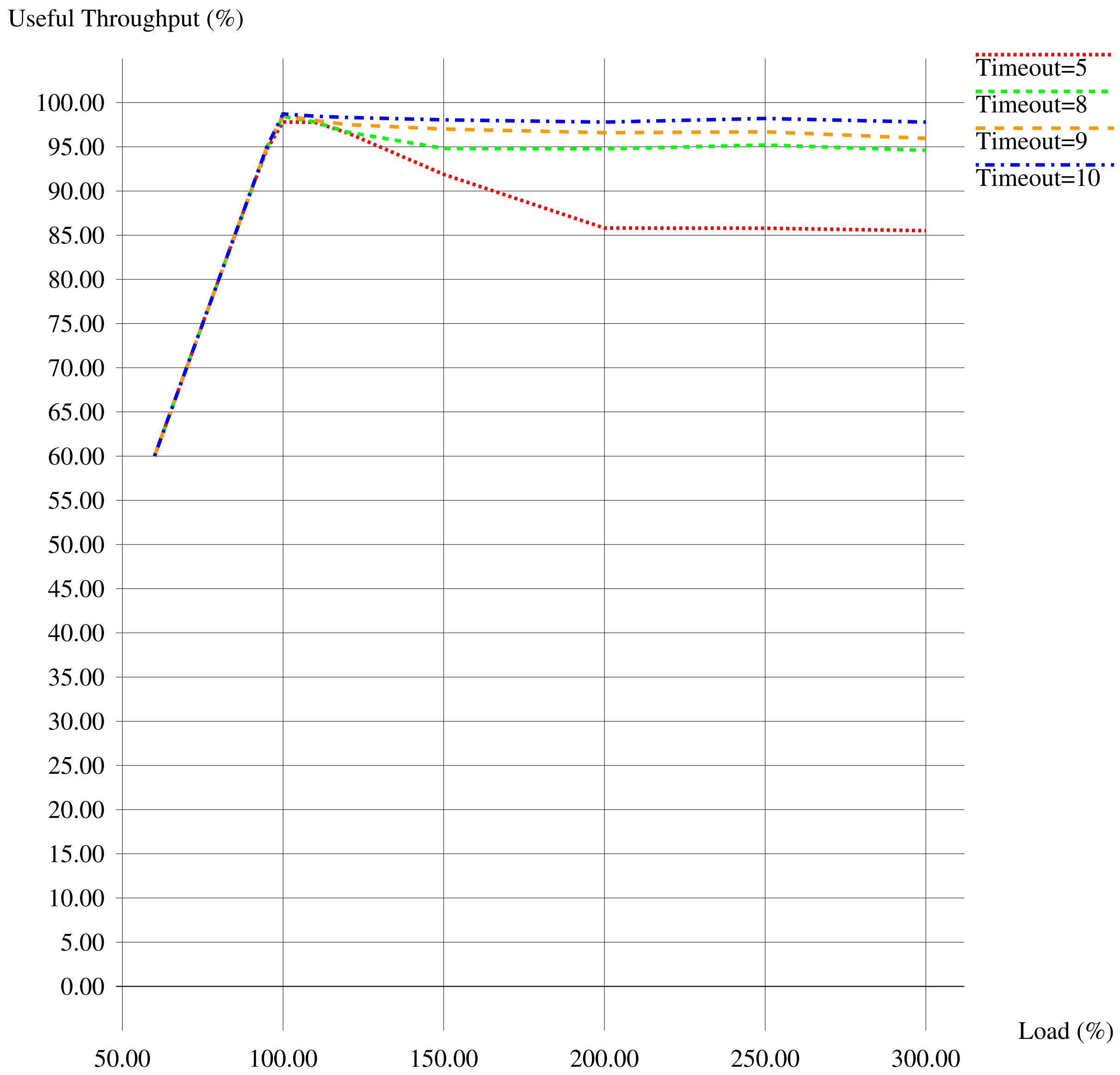

Figure 3: Throughput for a Server Augmented with the DTOC Strategy with Clients with No Retry.

the same client model as in Figure 2. The percentage of successfully-completed requests under high load is between 95% and 98% for client timeouts of 8, 9, and 10 seconds. For a client timeout of 5 seconds (which led to a server livelock situation under the regular strategy), the server throughput is around 85%, thus demonstrating DTOC's effective protection against livelock conditions.

One obvious question to ponder is why the useful server utilization is ever less than 100% when DTOC is enabled. There are three situations during which a server with DTOC is not performing useful work:

1. The server is idle with its listen queue empty.

2. The server is performing the DTOC check for a request.

3. The server is preparing a response to a request that passed the DTOC check, but that times out before the response preparation is complete.

Let us consider the case of a 300% load with a client timeout of 5 seconds, where we observed a 15% degradation from a 100% useful throughput level. Clearly at the 300% overload level, the server's listen queue is effectively never empty, so situation 1 is not an important factor. Also, the DTOC check overhead of situation 2 is minimal, with an upper bound of $300\% \cdot 0.5\% = 1.5\%$. In practice, this overhead will be even smaller because requests that encounter a full listen queue will be denied initial entry and will never be DTOC-checked. Thus, the requests that timeout during response preparation, as mentioned in situation 3 above, account for much the 15% degradation in useful server throughput seen for this case. This is further substantiated for a slightly different case in Figure 8.

One final question to consider with Figure 3 is why the throughput for the 5 second timeout case levels off at the 200% load point, and is flat for higher loads. First remember that a request that enters a nearly full listen queue might have to wait 1025 service times before its response comes back. This would take an average of 10.25 seconds if none of the requests have timed out. Clearly this cannot be the case here, since then all of the 5-second-timeout requests would time out. A more self-consistent hypothesis for this case is that 50% of the requests time out, and that a request that enters a nearly-full listen queue sees only about 512 actual service times before its response comes back. If the processing of only 50% of the requests in the queue keeps the server fully utilized, it must be a 200% load that actually enters the server's listen queue. Any additional request arrivals above the 200% load level would be denied entry to the server's full listen queue. Thus, this excess applied load is merely shed without any server impact and the server's throughput remains constant above the 200% level.

Moving on, Figure 4 shows the performance of a web server using the regular strategy, but now with the clients issuing one retry after a timeout. If a response is not received within the predetermined timeout period, the client closes its side of the client-server connection and re-issues the request one more time. Performance results shown in Figure 4 are very similar, although slightly worse, to the results in Figure 2.

Figure 5 shows the performance of a server using the DTOC strategy and with the same 1-retry client model as in Figure 3. The results clearly show an improved server performance compared to the corresponding results under the regular strategy. However, the performance is worse than that shown in Figure 3 for a server augmented with the DTOC strategy and clients with no retries.

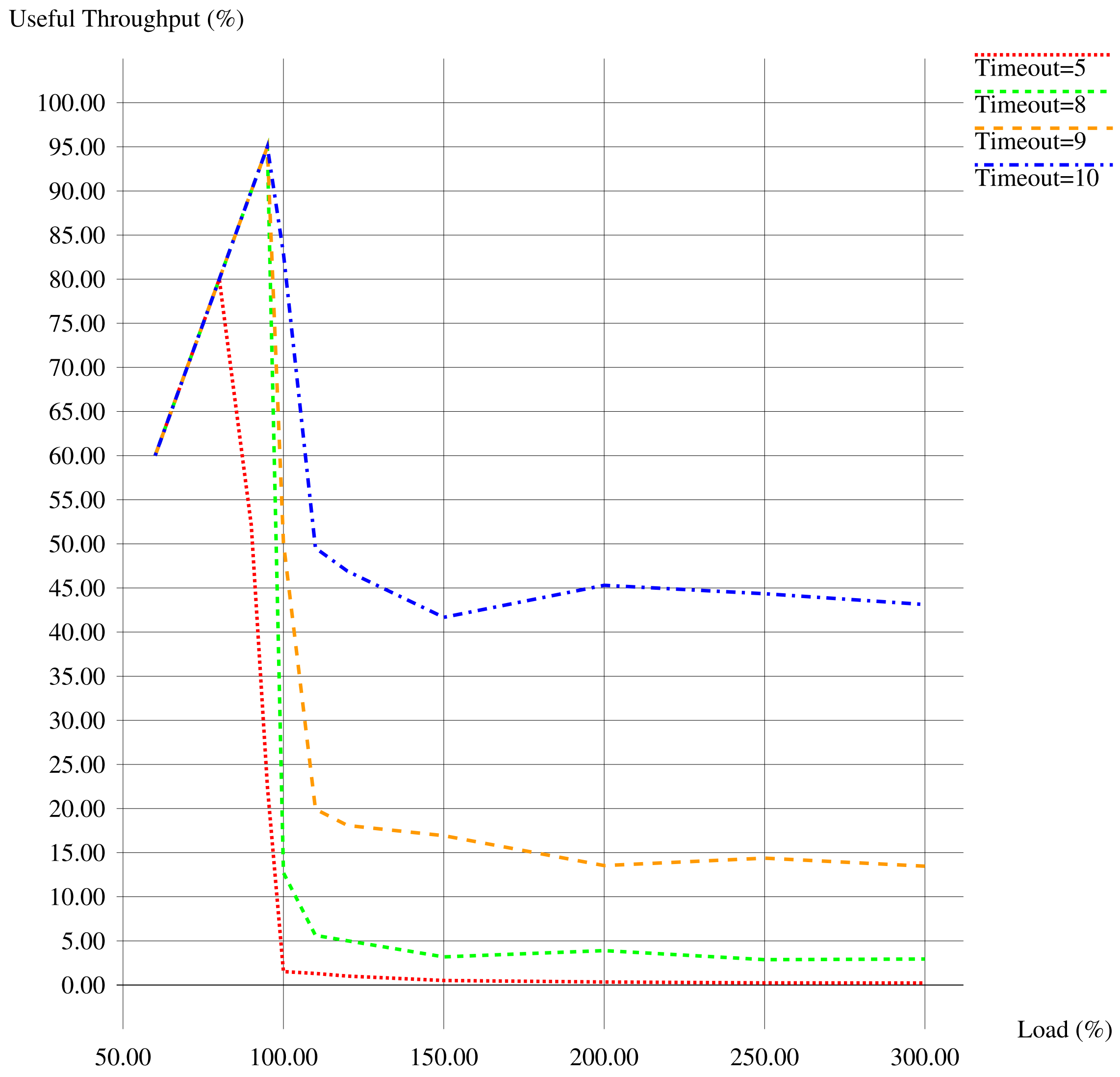

Figure 4: Throughput for a Server using the Regular Strategy with Clients with One Retry.

A detailed explanation of the results of Figure 5 is very instructive in understanding the more subtle implications of the client retry behavior. In particular, the following three distinguishing features of Figure 5 as compared to Figure 3 are worthy of further explanation:

1. **The useful throughput degrades much quicker for excess loads.** For example, the low-point of throughput for the 5-second timeout case is achieved at a load of 150% for the 1-retry case (Figure 5), whereas the low-point is achieved at a 200% load for the no-retry case (Figure 3). This demonstrates the additional load that the retry traffic generates. If 50% of the initial request load gets resubmitted as retries, then a 150% load with retries is comparable to a 225% load with no retries.

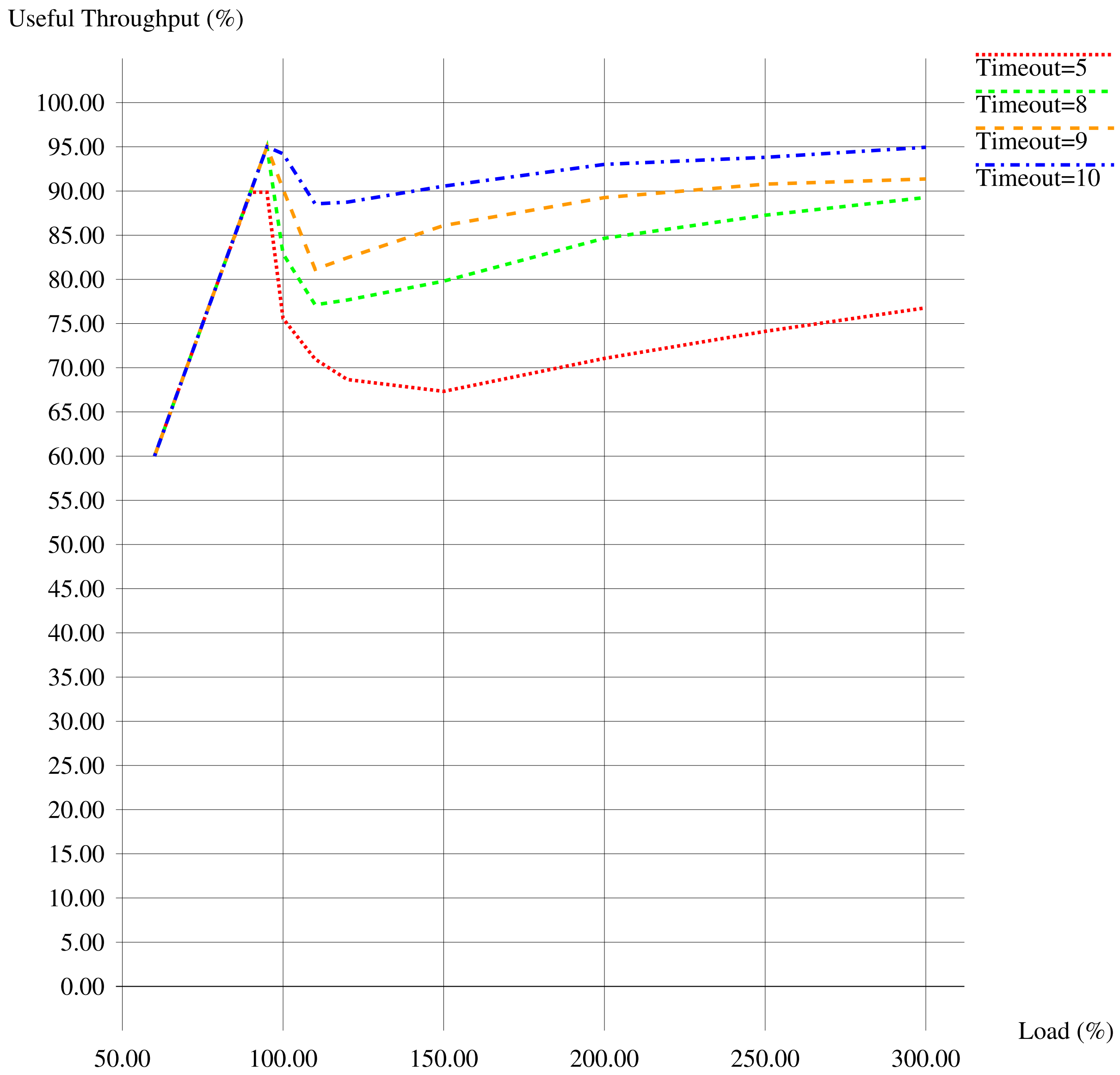

Figure 5: Throughput for a Server Augmented with the DTOC Strategy with Clients with One Retry.

2. **The useful throughput degrades to a lower level for excess loads.** Remember that the throughput degradation is caused primarily by requests that pass the DTOC check, but that timeout during their servicing. Requests for large files play a major role in this degradation, since the larger-file requests have a higher probability of timing out during their servicing than do the smaller files. The reason for this is simple- whereas both large and small requests have the same average in-queue wait, the large requests have a much larger in-service wait and hence timeout more during this stage. If more large requests timeout during servicing, one would expect that a smaller percentage of the large requests are successfully processed. With these large

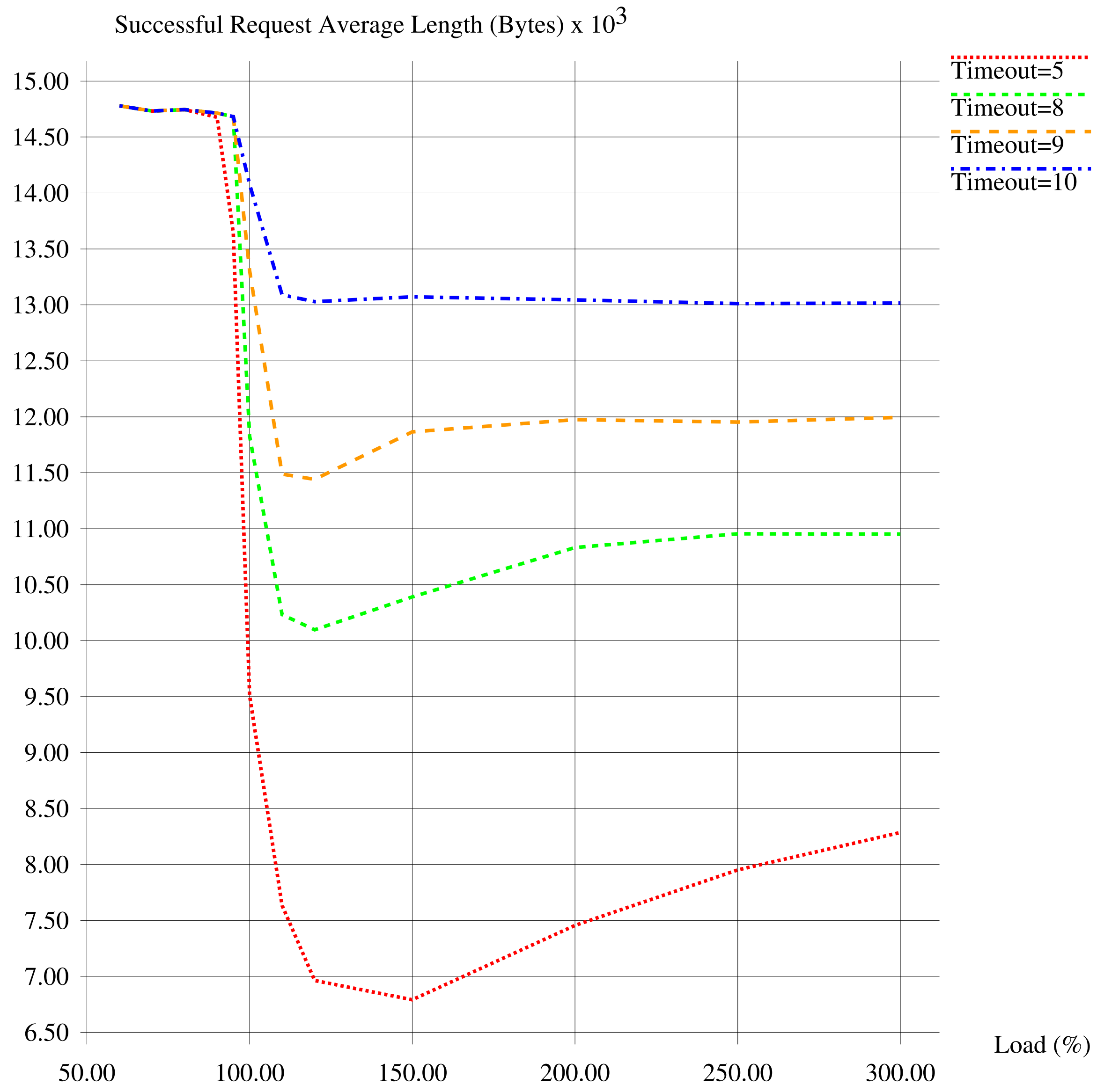

Figure 6: Average Size of Successfully Completed Requests for a Server Augmented with the DTOC Strategy with Clients with One Retry.

requests removed from the mix of successfully-completed requests, the average filesize of the successfully-completed requests should fall for excessive loads. This we observed, as shown in Figure 6. The average file size of the input mix is 14675 bytes, whereas the average file size for successfully-completed requests for loads above 100% is less in all cases. Another representation of this same effect is Figure 7, which displays the distribution of successfully-completed requests by percentage for the different file classes. It shows that an overloaded server indeed discriminates against the larger files. The output distribution is skewed in favor of requests from the small-file classes 0 and 1, whereas the percentage of successfully-completed requests from the large-file classes 2 and 3 is much lower than in the original input mix.

The retried-request stream thus effectively "pollutes" the distribution of files that the server sees with large files. Since the large files timeout more often during their processing, it is this large-request pollution that causes the additional throughput degradation seen in Figure 5 vs. Figure 3.

3. **The useful throughput exhibits a recovery for very high loads.** Note that the stream of timed-out requests must compete with the new-request arrivals for openings in the server's queue. For highly excessive loads, the new requests will quickly fill any openings in the queue, and requests that timeout will not be able to re-enter the queue as retries. A high applied load will thus suppress any effects of the request retries. The server will see a request filesize distribution unpolluted by any retry stream and will achieve throughputs closer to the no-retry case. Figure 5 does indeed show performance results that climb back toward the no-retry levels shown in Figure 3.

Since we have demonstrated that the successfully-processed file mix is no longer the same as the input mix, we should caution the reader not to view the %-throughput performance numbers as a %-utilization. Thus, in an overload situation, an 80% useful throughput rating *does not* imply that the server is busy 80% of the time processing successfully-completed requests. Rather, it implies that the server is processing 80 requests/sec, which is 80% of the 100 requests/sec rate the server can achieve on an unadulterated input file mix. Note that, under this definition, sustained throughputs greater than 100% are theoretically possible.

Finally, in support of our claims about the source of the server's throughput degradation, Figure 8 shows the percentage of requests that time out during processing for a server using the DTOC strategy with 1-retry clients. For a timeout of 5 seconds, the timeout percentage is significant: 14% of all requests. This shows a potential for future improvements: if the requests which time out during their processing are aborted as soon as they time out, it might boost the server performance even further. Earlier, in presenting implementation options for DTOC, we discussed a more signal-driven approach which would asynchronously notify the server when a client closed its client-server connection (i.e. timed out). This type of approach is worth further exploration as a way of recovering perhaps 50% of the server throughput degradation caused by these in-service client timeouts.

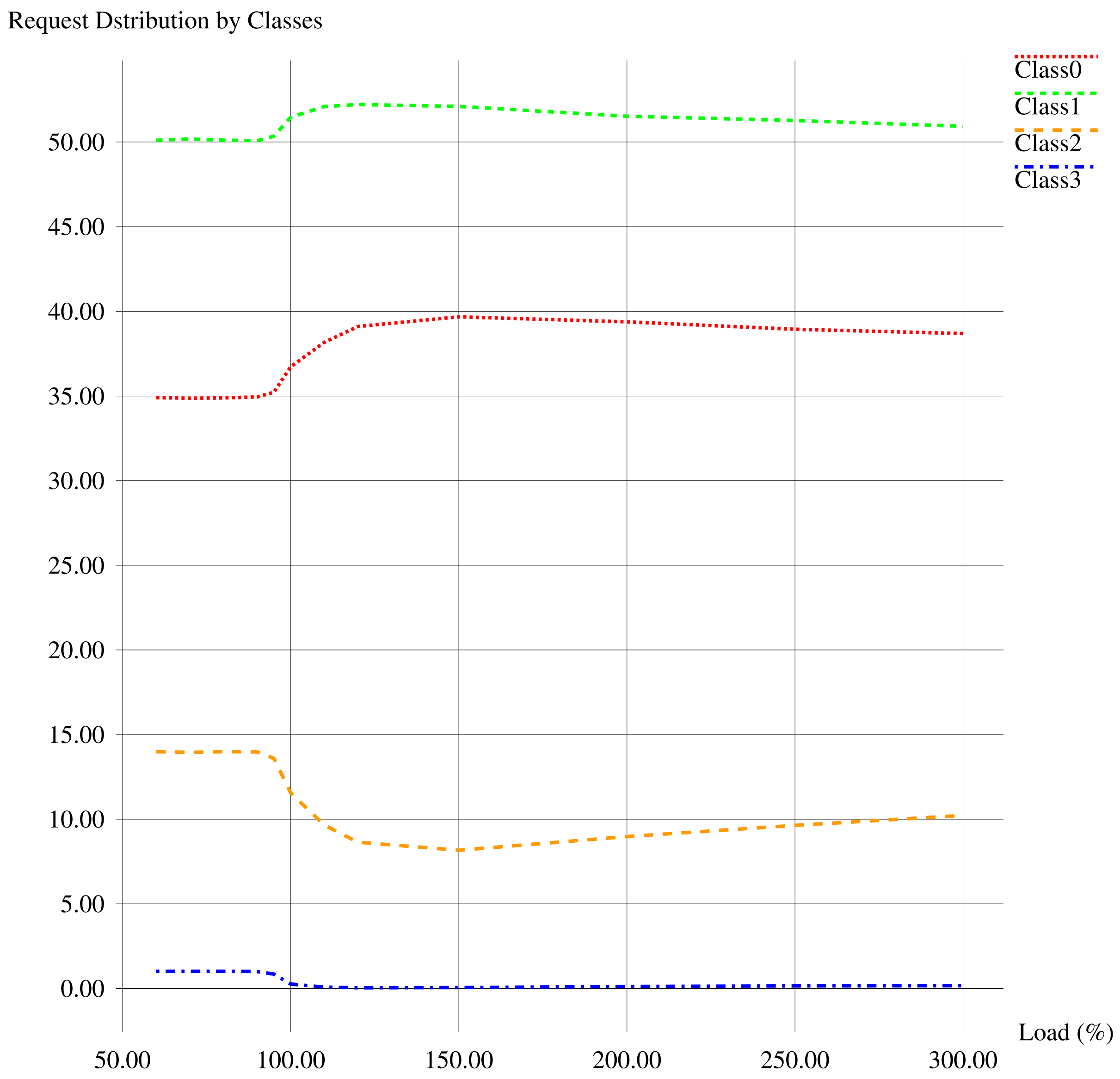

Figure 7: Class Distribution of Successfully Completed Requests for a Server Augmented with the DTOC Strategy with Clients with One Retry.

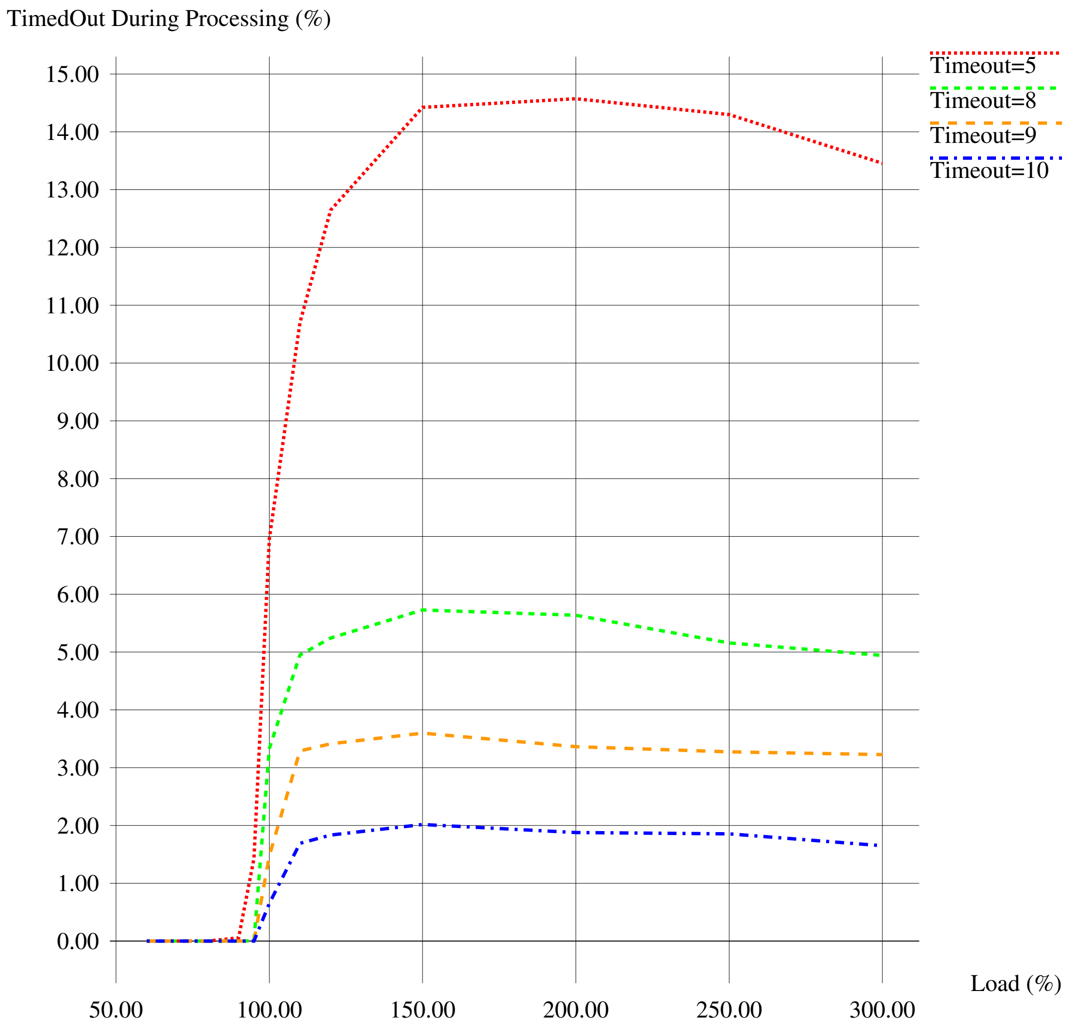

Figure 8: Percentage of Requests Accepted for Processing but Timed Out during that Processing.

The analysis presented in this section convincingly shows that a web server the using DTOC strategy achieves significantly higher performance results under excess loads than a web server using the regular strategy. Moreover, it was shown that a web server which encounters clients with relatively short timeouts can easily enter a request-timeout livelock state under only slightly-overloaded conditions. This same problem exists for a server with a longer listen queue and proportionally longer client timeouts. In addition, it was shown that a server augmented with the DTOC strategy easily "escapes" this livelock situation and demonstrates excellent throughput as measured in the number of successfully-completed requests/sec. This new metric is a crucial one to adopt in order to distinguish truly-useful server work. It allows one to clearly identify the server livelock state as well as to analyze the overall amount of wasted server resources.

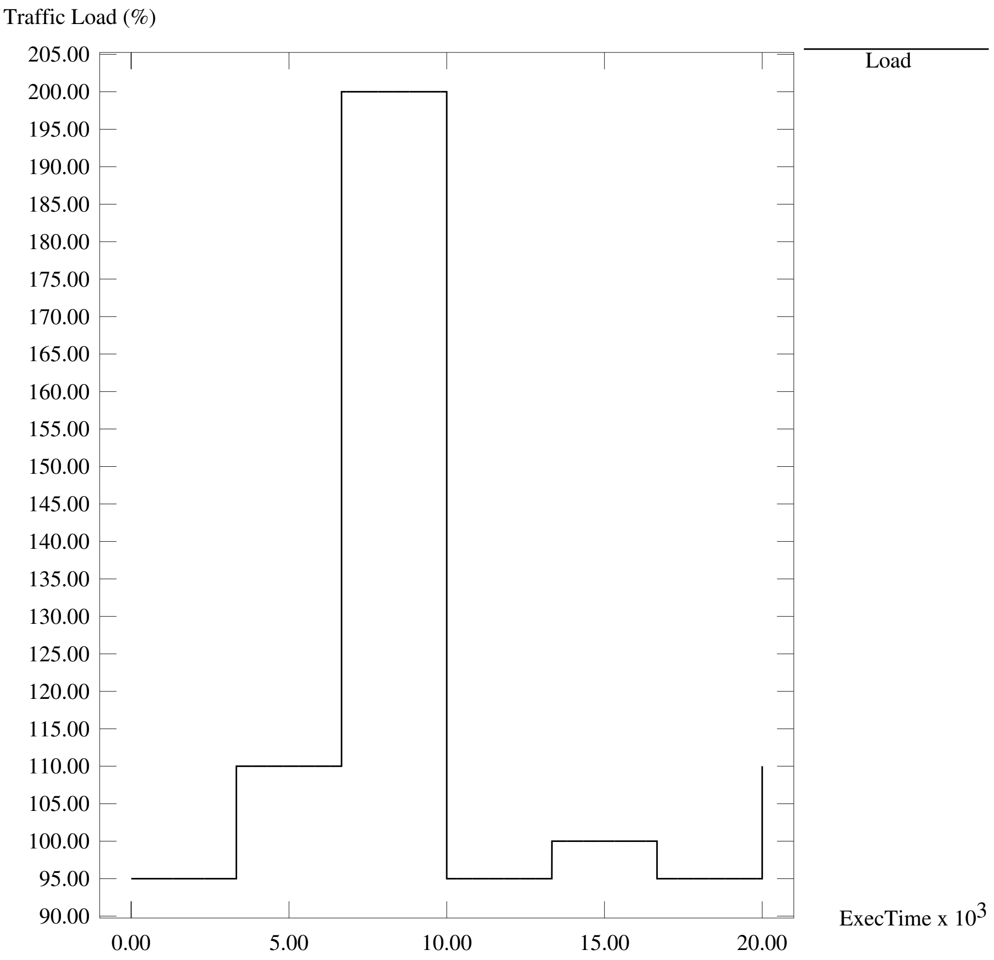

Figure 9: "Normal Day" Workload Traffic Pattern.

4.2 Using the DTOC Strategy for Improving Server Performance under Variable Load Traffic Patterns

The newly-proposed DTOC strategy is designed to make a web server more efficient during periods of excessive load. We designed two variable traffic patterns to verify whether the DTOC strategy consistently improves server performance and adequately adjusts its behavior depending on traffic rates.

The first traffic pattern is defined by the pattern showed in Figure 9. We call it the *"normal day"* workload traffic pattern. This traffic pattern specifies a moderate overload during 33% of the day, and the rest of the time, it specifies a load just under the server's capacity. We conjecture that this type of load is typical in practice: most of the time, the load is manageable, and only for a few peak periods is it high.

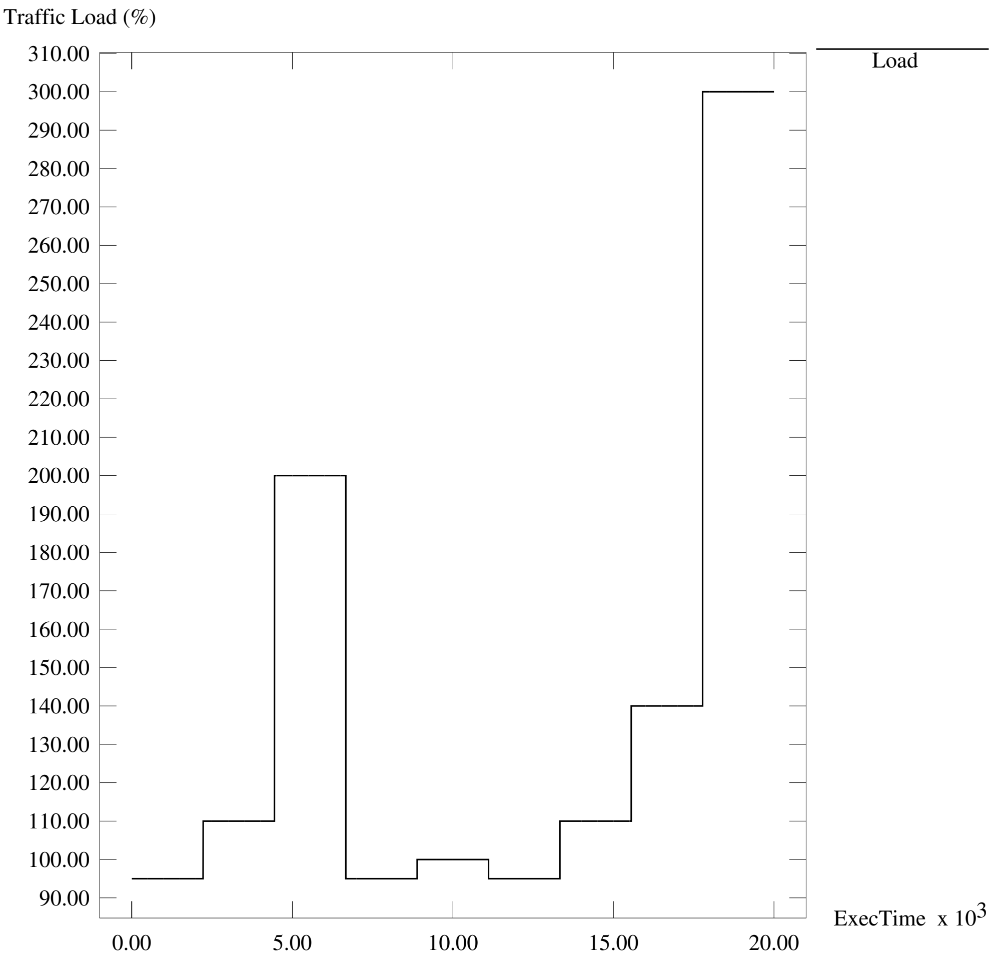

Figure 10: "Busy Day" Workload Traffic Pattern.

The second workload is defined by the pattern showed in Figure 10. We call it the *"busy day"* workload. This traffic pattern specifies an overload during 56% of the day, including a heavy overload interval of 300%. The remainder of the time, as in the first traffic pattern, is spent with a load just under the server's capacity.

We did not simulate what might be called a *"bad day"* workload (with consistently high overload for all intervals) since the results are predictable. We will comment more on this at the end of this section.

Figure 11 shows the useful server throughput for a *"normal day"* traffic pattern, comparing the case of a server using the regular strategy against one where the server is augmented with the DTOC strategy. In both cases, the clients perform 1 retry if necessary. As in past graphs, the throughput is expressed as the rate of successfully-completed requests, normalized to the server capacity.

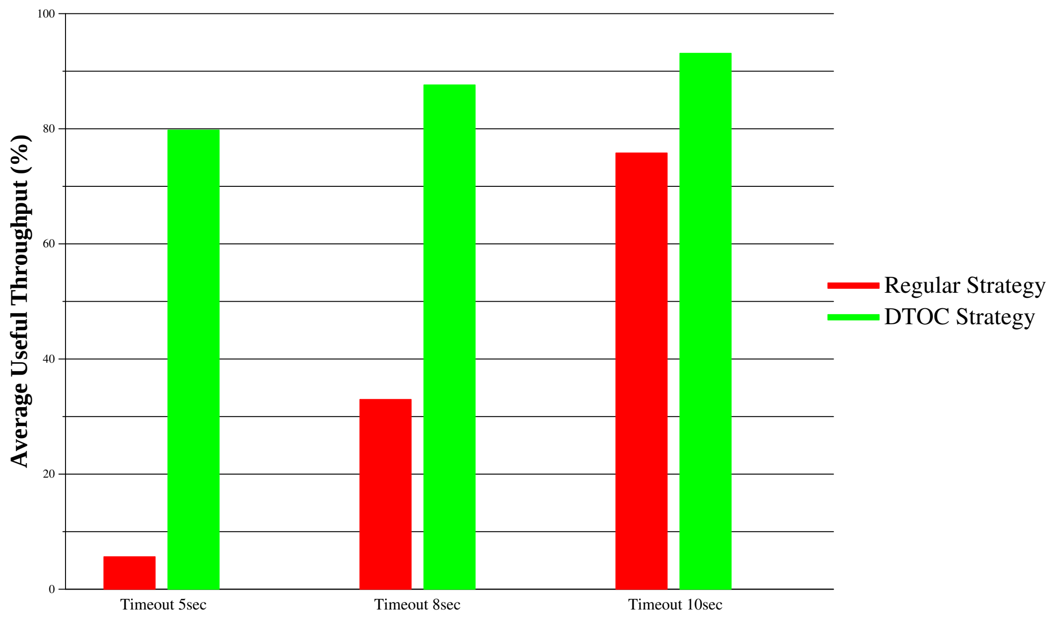

Figure 11: Throughput for a Server using the Regular Strategy versus a Server Augmented with the DTOC Strategy with 1-Retry Clients and Different Timeout Values for a "Normal Day" Workload.

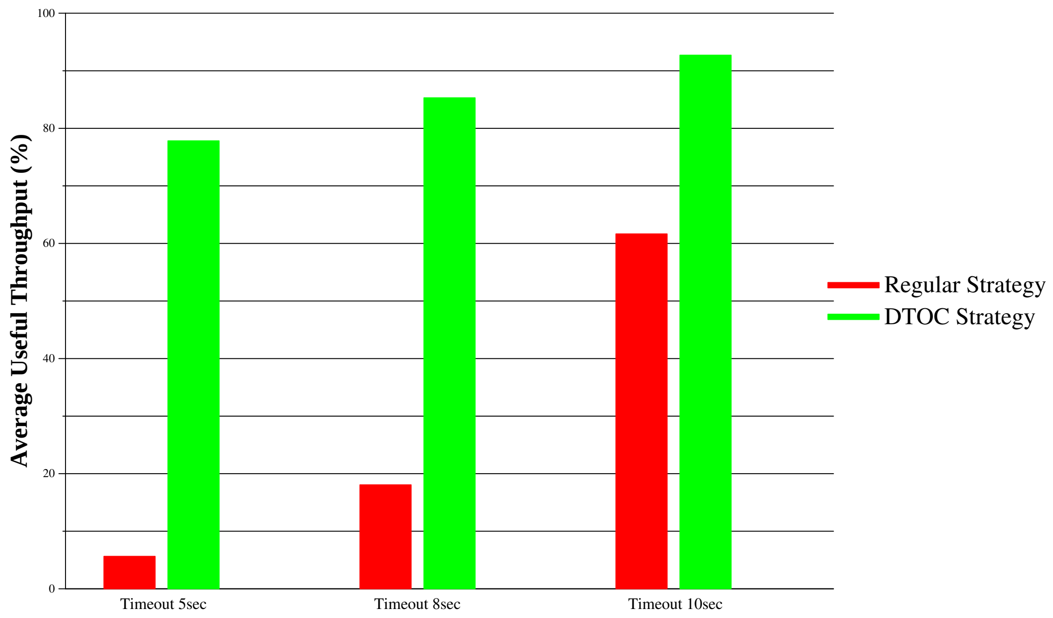

Figure 12: Throughput for a Server using the Regular Strategy versus a Server Augmented with the DTOC Strategy with 1-Retry Clients and Different Timeout Values for a "Busy Day" Workload.

For a model with a client timeout of 5 seconds, the server, using the regular strategy, soon enters a request-timeout livelock state from which it cannot recover. However, using the DTOC strategy, the server achieves a throughput of 80%, thus demonstrating DTOC's effective protection against livelock conditions.

If we increase the modeled client timeout to 8 seconds, the server, using the regular strategy, provides an improved throughput of 33%. This throughput is achieved in spite of the server periodically entering and recovering from request-timeout livelock. The same server, using the DTOC strategy, achieves a throughput of 88%, which is 2.7 times the performance of the regular strategy case.

Even for a model with a client timeout of 10 seconds, the server augmented with the DTOC strategy still provides a throughput of 93% as compared against a 76% throughput under the regular strategy. This represents a 22% server performance improvement.

Figure 12 shows the server throughput in successfully-completed requests for a *"busy day"* traffic patterns, comparing a server that employs the regular strategy against one augmented with the DTOC strategy. For this case, the performance comparison shows an even stronger benefit in using the DTOC strategy.

For a model with a client timeout of 5 seconds, the server, using the regular strategy, predictably enters a request-timeout livelock and can not recover from it. Meanwhile a server, using the DTOC strategy, achieves a throughput of 78%, once again demonstrating DTOC's effectiveness against livelock conditions.

For a model a with client timeout of 8 seconds, the server, using the regular strategy, provides a throughput of 18%, whereas the server, using the DTOC strategy, achieves an 85% throughput. This represents a 4.7 times performance improvement of DTOC against the regular strategy case.

Finally, for a model with a client timeout of 10 seconds, the server using the DTOC strategy provides a throughput of 93%, as compared against 62% under the regular strategy. This represents a 50% server performance improvement.

We have found that graphing the server throughput against the time yields a number of insights. Figure 13 shows the percentage of successfully completed requests in 10 seconds intervals for the case of a "normal day" workload with a client timeout of 5 seconds. This clearly illustrates how a server, using the regular strategy enters a request-timeout livelock at the 1,200 second mark of the 20,000 second run, and subsequently cannot recover. The server ends up processing only timed-out requests and, as a result produces no useful work. This emphasizes the importance of adequate metrics and analysis for measuring a web server's performance.

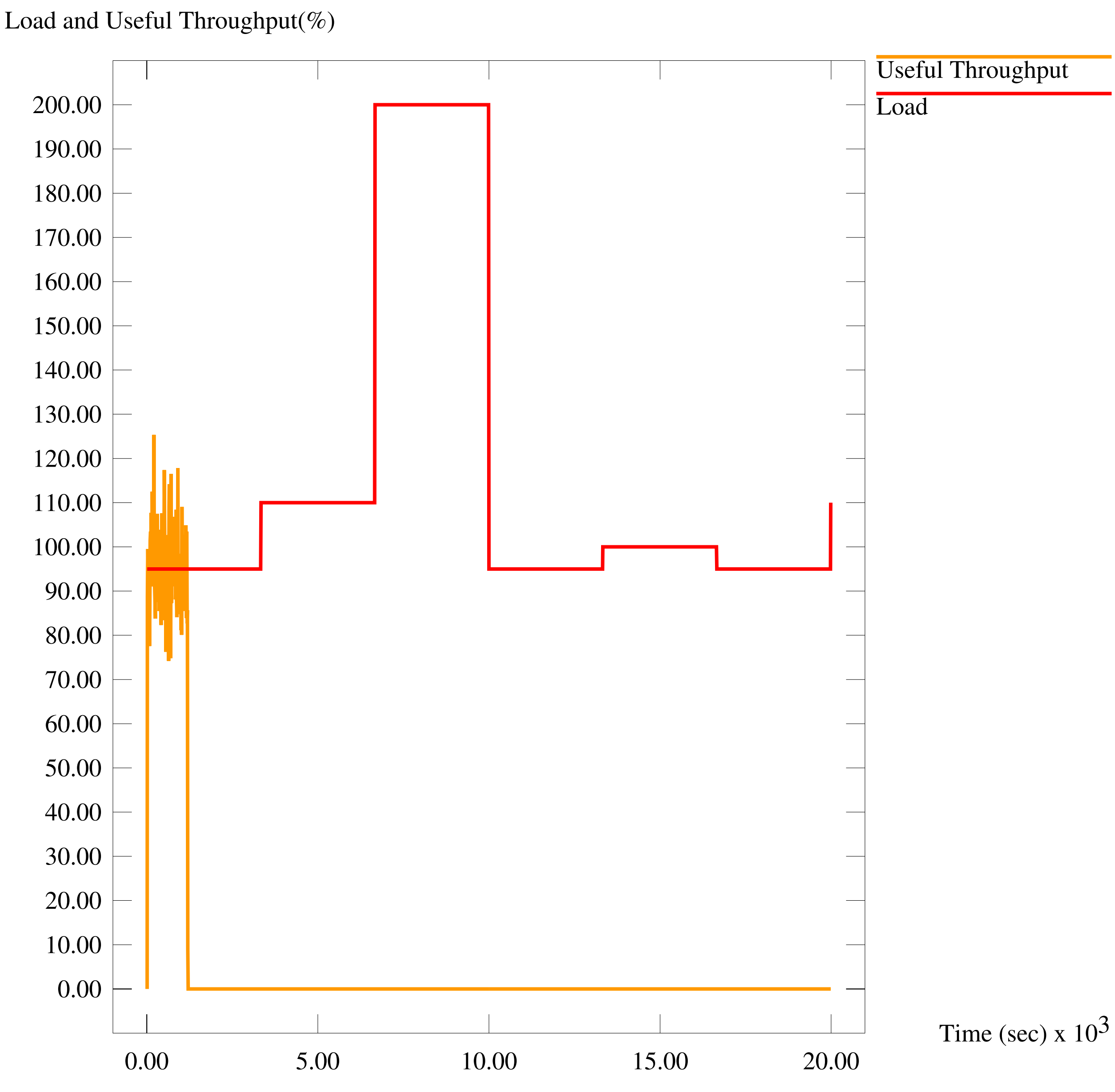

Figure 13: Server Livelock under the Regular Strategy for a "Normal Day" Workload Traffic Pattern with 1-Retry Clients with a 5 second Timeout.

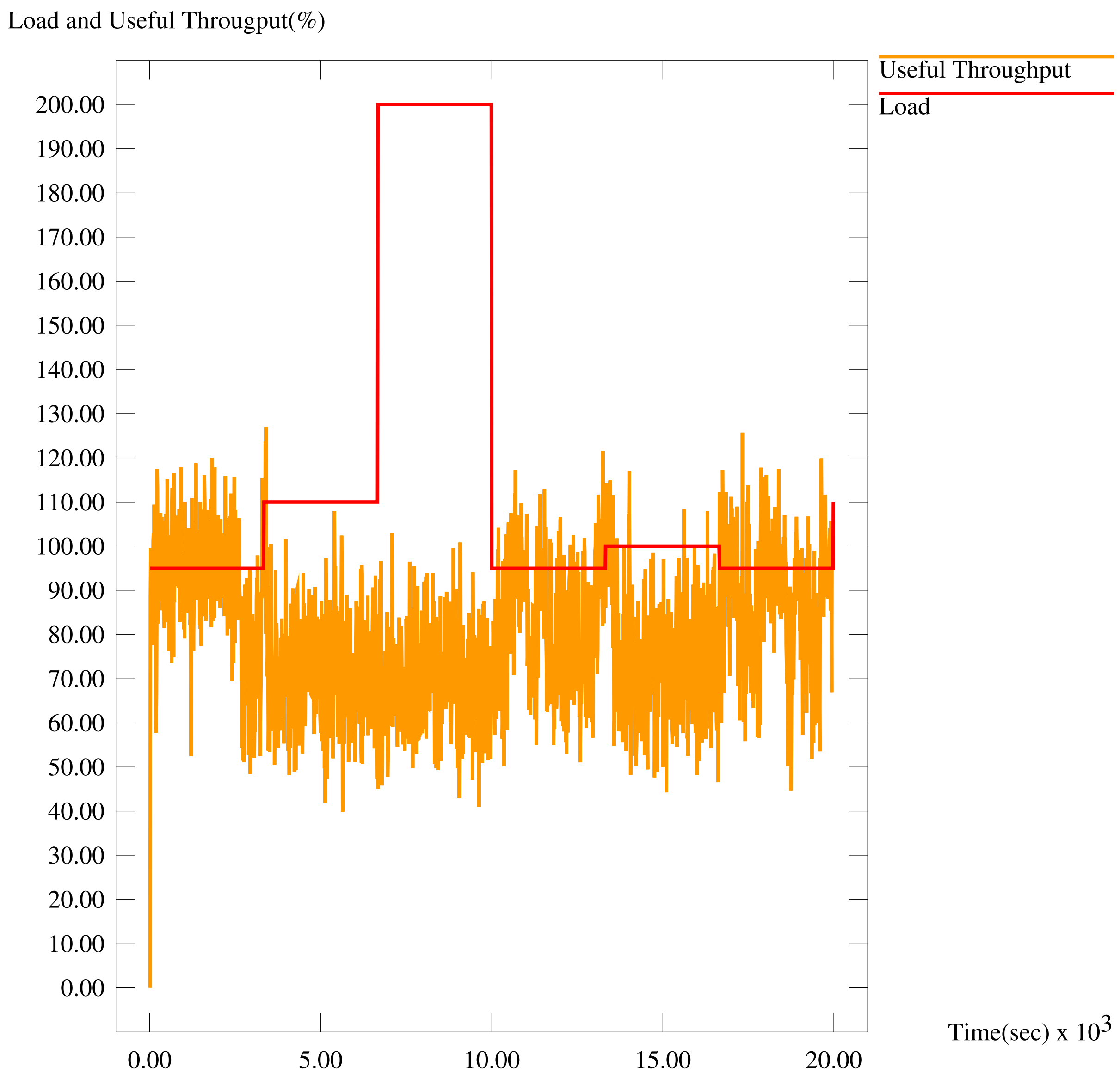

Figure 14: Server Livelock Avoidance with the DTOC Strategy for a “Normal Day” Workload with 1-Retry Clients with a 5 second Timeout.

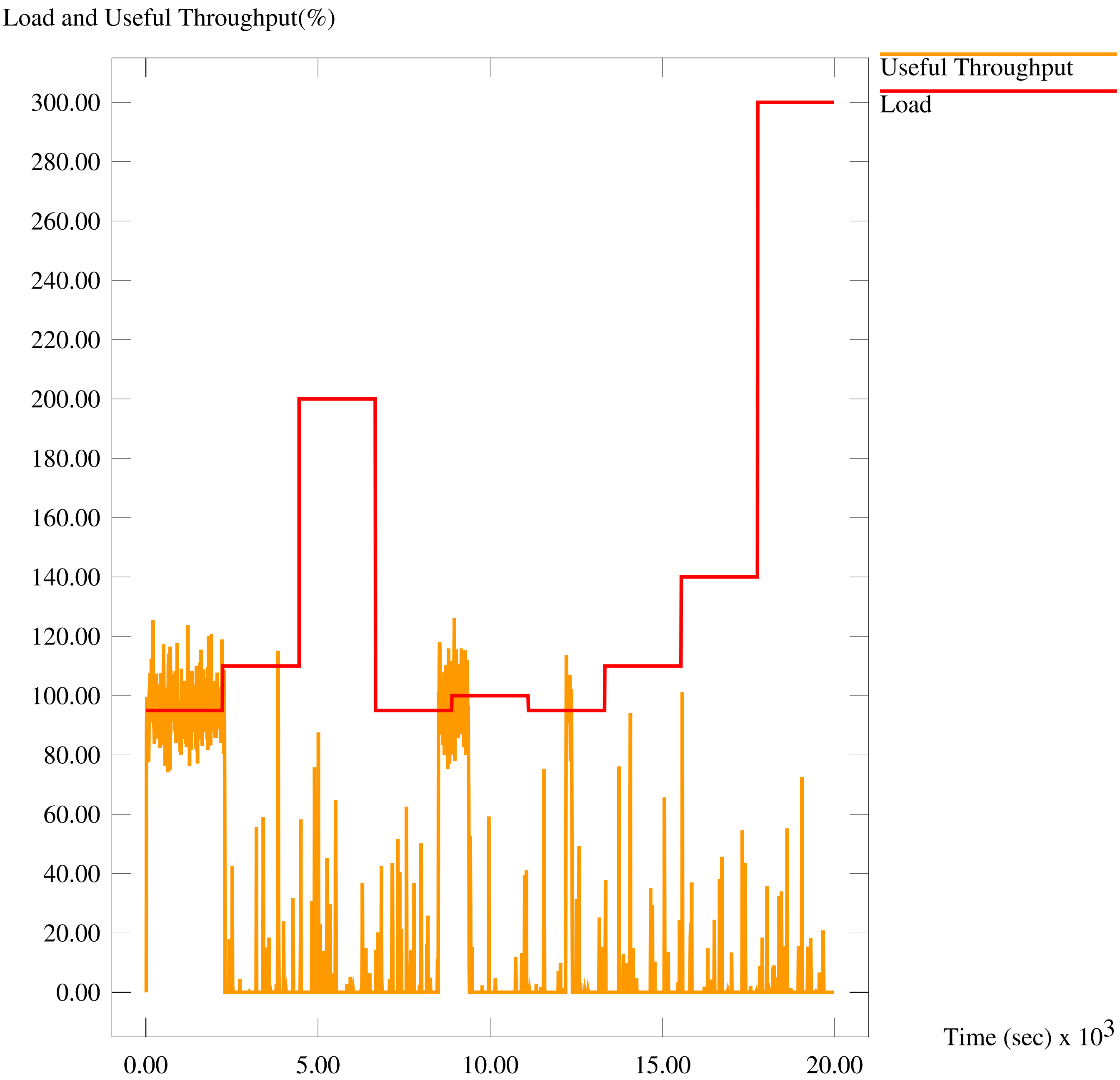

Figure 15: Poor Server Performance under the Regular Strategy for a "Busy Day" Workload with 1-Retry Clients with an 8 second Timeout.

Figure 15 shows the server behavior under the regular strategy while it was executing the "busy day" workload with a client timeout of 8 seconds. It illustrates how the server can periodically enter a request-timeout livelock, but eventually can recover from it. Overall, this results in very poor overall server performance: the useful throughput is only 18%, at a time when the server utilization approaches 100%.

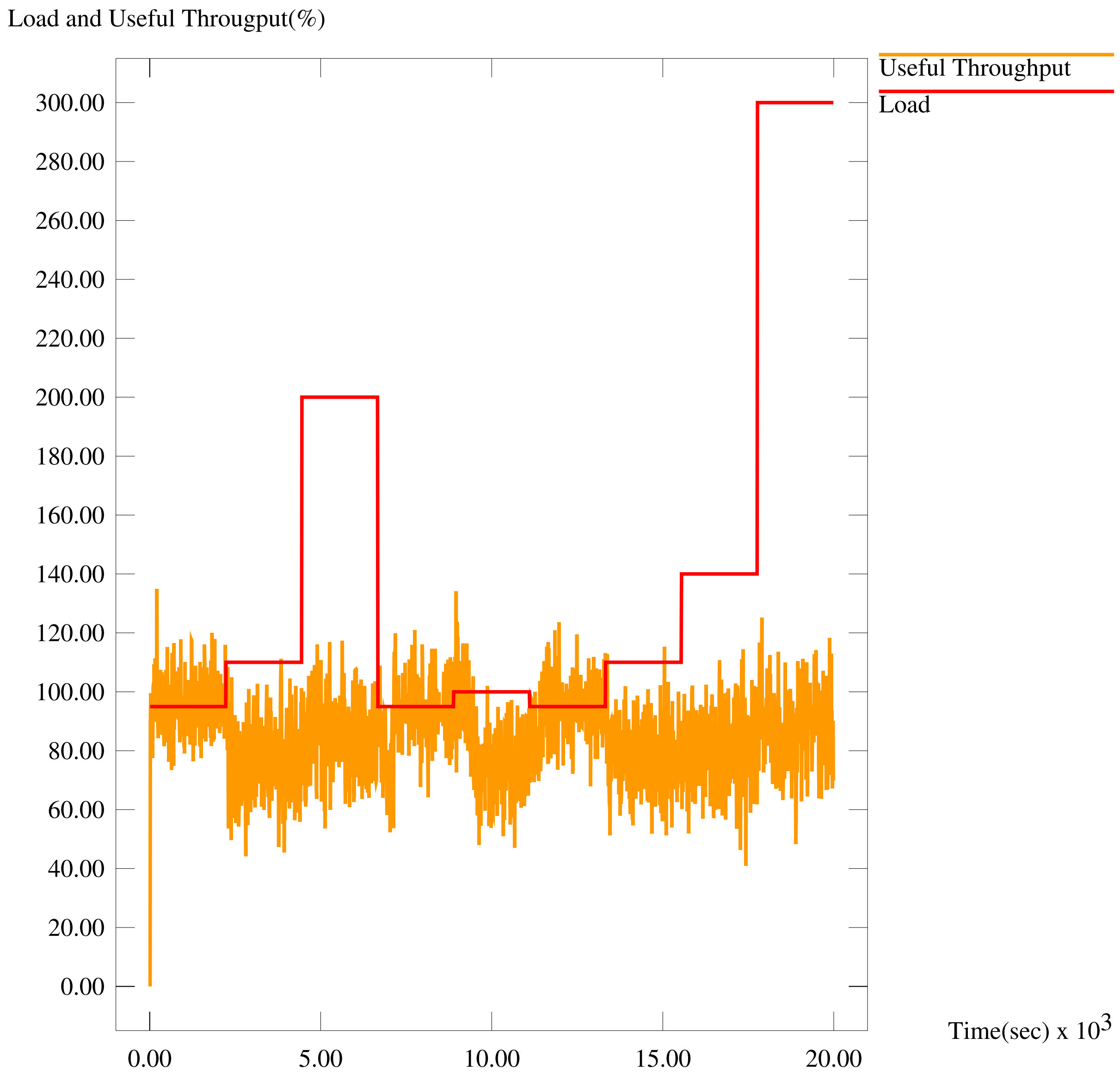

Figure 16: Server Performance with the DTOC Strategy for a "Busy Day" Workload with 1-Retry Clients with an 8 second Timeout.

Finally, Figure 16 shows the improved behavior of a server using DTOC strategy: note the consistently high throughput across all the points of the workload.

From this analysis, we can project the results for what might be called a "bad day" workload that has a high overload for all intervals. Clearly, loads consistently worse than those described will only worsen the performance and chances of livelock for a web server that uses the regular strategy. The DTOC strategy allows the server, even in periods of consistent overload, to avoid livelock and to significantly improve useful server throughput.

5 Analytical Model

Analytical models can sometimes be used to corroborate the results of simulation results, although it is more-often the case that the system simulated is too complex for a closed-form analytical solution. Luckily, the simplest of our simulations, that of a server with no DTOC and no client-retry, is within the realm of analytical methods. We now present the supporting theory for the simulation results of Figure 2 along with some additional insights that this analysis brings.

5.1 Deriviation of the Throughput formula for a M/M/1/K model

In this section, we propose to derive the equation that defines a server's normalized "useful" throughput X_{useful} as a function of the normalized offered load a, the system capacity K and the client timeout T_{client}. This is exactly what was simulated in Figure 2 for the case of $K = 1025$ and for T_{client} values of 5, 8, 9 and 10 seconds. We derive this relationship for a system that is based on an $M/M/K/1$ queueing model, that is to say, we assume:

- arriving customers (hereafter called requests) arrive at an average rate of λ and with exponentially-distributed interarrival times;
- requests are serviced at an average rate μ with exponentially-distributed service times having a mean $W_s = 1/\mu$;
- the system has a total capacity of K requests (and hence a queue of length $K - 1$);
- the system has 1 server.

Note that these assumptions depart from our simulated model as follows:

- the simulation model assumes a server capacity of 100 requests/sec, a system capacity K of 1025 and various specific client timeouts, whereas we treat these quantities with the variable parameters μ, K and T_{client} here;
- the simulation has a service time distribution based on a SpecWeb-like model, whereas the analytical model assumes an exponential distribution. More will be said on this difference when we compare the analytical and simulation results.

Note that in Figure 2, both axes have been normalized to (i.e. divided by) the assumed server capacity of 100 requests/sec. The fact that we treat the server throughput with the general parameter μ in the equations that follow is not of great significance, since when we go to graph these equations, we too will normalize the throughput and load axes by dividing by μ.

Note, however, that we will not scale the throughput and load by a factor of 100 in order to create percentages, as was done in the simulation graphs.

Let us now embark on deriving the equation for the server's normalized throughput X_{useful} as a function of the normalized offered load $a = \lambda W_s = \lambda/\mu$ and also the parameters K and T_{client}. We do this by building on the analysis offered by Arnold O. Allen in [A90]. We repeat the following results, taken from Section 5.2.2 of this book, on the M/M/1/K Queueing System:

On page 270 of [A90], eqation 5.68 gives the steady-state probabilities for finding n requests in the system at any random time (as a function of K and a) for $n = 0, 1, \ldots, K$:

$$p_n = \begin{cases} \frac{(1-a)\ a^n}{1-a^{K+1}} & \text{for } a = \frac{\lambda}{\mu} \neq 1 \\ \\ \frac{1}{K+1} & \text{for } a = \frac{\lambda}{\mu} = 1 \end{cases} \tag{1}$$

Allen also gives the probability that an arriving request *that successfully enters the system* finds n requests already in the system. This is given on page 272 of [A90], equation 5.78 as:

$$q_n = \frac{p_n}{1-p_K} \qquad n = 0, 1, \ldots, K-1. \tag{2}$$

Finally, Allen derives the cumulative distribution function $W[t]$ for the total wait time of requests that successfully enter the system. This wait time includes both in-queue and in-service waiting for a system in the steady state. If w is a random variable describing the total wait time of a request that enters the system, then equation 5.79 from Allen's book is presented as:

$$P[w <= t] = W[t] = 1 - \sum_{n=0}^{K-1} q_n Q[n; \mu t] \tag{3}$$

with $Q[n; \mu t]$ given by equation 5.80 as

$$Q[n; \mu t] = e^{-\mu t} \sum_{i=0}^{n} \frac{(\mu t)^i}{i!} \tag{4}$$

We can now state, in its most intuitive form, the equation for $X_{useful}(a, K, T_{client})$ that we seek. We realize that under the most optimistic conditions, the server's useful throughput can be at most equal to the offered load. However, we must factor out those requests that arrive, only to find the server's queue full. In addition, for those requests that successfully enter the server queue, we must factor out those requests that timeout before their service is complete. Thus, we have:

$$\begin{aligned} X_{useful}(a, K, T_{client}) &= \\ a \cdot P[request\_enters\_the\_system] &\cdot P[request\_wait\_time <= T_{client}] \end{aligned} \tag{5}$$

A request can enter the system if it does not find K requests already in the system, so that

$$P[request\_enters\_the\_system] = 1 - p_K$$

and as previously presented, for requests that successfully enter the system,

$$P[request\_wait\_time <= T_{client}] = W[T_{client}]$$

Now plugging these last 2 results into equation 5 above, we get

$$X_{useful}(a, K, T_{client}) \; = a \cdot (1 - p_K) \cdot W[T_{client}]$$

Expanding $W[T_{client}]$ using equation 3 yields

$$X_{useful}(a, K, T_{client}) \; = a \cdot (1 - p_K) \cdot \left(1 - \sum_{n=0}^{K-1} q_n Q[n; \mu T_{client}]\right)$$

Now plugging in the value for q_n from equation 2

$$X_{useful}(a, K, T_{client}) \; = a \cdot (1 - p_K) \cdot \left(1 - \sum_{n=0}^{K-1} \frac{p_n}{1 - p_K} Q[n; \mu T_{client}]\right)$$

Which simplifies to

$$X_{useful}(a, K, T_{client}) \; = a \cdot \left(1 - p_K - \sum_{n=0}^{K-1} p_n Q[n; \mu T_{client}]\right)$$

Plugging in Allen's result for $Q[n; \mu t]$ from equation 4,

$$X_{useful}(a, K, T_{client}) \; = a \cdot \left[1 - p_K - \sum_{n=0}^{K-1} p_n e^{-\mu T_{client}} \sum_{i=0}^{n} \frac{(\mu T_{client})^i}{i!}\right] \tag{6}$$

Noticing that T_{client} is always multiplied by μ in the previous equation, we define a new normalized form of the client timeout that is measured, not in seconds, but in a unit of time equal to the average service time W_s, that is to say

$$T_{norm} = T_{client}/W_s = \mu T_{client}$$

Adapting equation 6 to this definition, we get

$$X_{useful}(a, K, T_{norm}) \; = a \cdot \left[1 - p_K - \sum_{n=0}^{K-1} p_n e^{-T_{norm}} \sum_{i=0}^{n} \frac{T_{norm}^{\,i}}{i!}\right] \tag{7}$$

where p_n and p_K are defined by equation 1. Making this last substitution, we get

$$X_{useful}(a, K, T_{norm}) \; = a \cdot \left[1 - \frac{(1-a)a^K}{1 - a^{K+1}} - \sum_{n=0}^{K-1} \frac{(1-a)a^n}{1 - a^{K+1}} e^{-T_{norm}} \sum_{i=0}^{n} \frac{T_{norm}^{\,i}}{i!}\right]$$

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}(1-a^{K+1}-(1-a)a^K - \sum_{n=0}^{K-1}(1-a)a^n e^{-T_{norm}}\sum_{i=0}^{n}\frac{T_{norm}^{\,i}}{i!})$$

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}(1-a^K - e^{-T_{norm}}\sum_{n=0}^{K-1}(1-a)a^n\sum_{i=0}^{n}\frac{T_{norm}^{\,i}}{i!}) \quad (8)$$

This then is the equation that we seek for $X_{useful}(a,K,T_{norm})$. We found though that this equation, with its 2 nested summations, evaluated quite slowly using *Mathematica*[1] on a 233MHz Kayak PC. Luckily, some simplifications were possible. Because the details are somewhat lengthly and distract from the thrust of this effort, we present them as Appendix A.1, and merely state the simplified result here:

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1-a^K - e^{-T_{norm}}\sum_{i=0}^{K-1}\frac{T_{norm}^i}{i\,!}(a^i-a^K)\right] \quad (9)$$

The single summation in the above equation can be hidden through the use of the *incomplete gamma function*, which, for integer K, takes the form

$$\Gamma(K,T) = (K-1)!\,e^{-T}\sum_{i=0}^{K-1}\frac{T^i}{i!}$$

We explore the use of $\Gamma(K,T)$ in Appendix A.1 as well, and merely present the transformed result here:

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1-a^K - e^{(a-1)T_{norm}}\frac{\Gamma(K,aT_{norm})}{(K-1)!} + a^K\frac{\Gamma(K,T_{norm})}{(K-1)!}\right]$$

Because *Mathematica* has built-in evaluators for the incomplete gamma function as describe in [W96], this last form proved the most efficient for getting prompt graphing of our throughput formula.

5.2 Comparison of Analytical and Simulation Results

We are now in a position to compare the results of our analytical work against the simulated results of Figure 2. Figure 17 shows a graph of $X_{useful}(a,K,T_{norm})$ for a system capacity that matches our simulations, namely $K = 1025$, and for T_{norm} values of 1000, 1025 and 1050.

Qualitatively, the analytical curves show a strong resemblance to the simulated results of Figure 2. In both cases, $X_{useful}(a,K,T_{norm}) = a$ for loads the server can handle (i.e. for $0 <= a <= 1$). In addition, for excessive loads, the useful throughput quickly falls to an asymptotic level less than 1.

[1] Wolfram Research, Champaign, IL

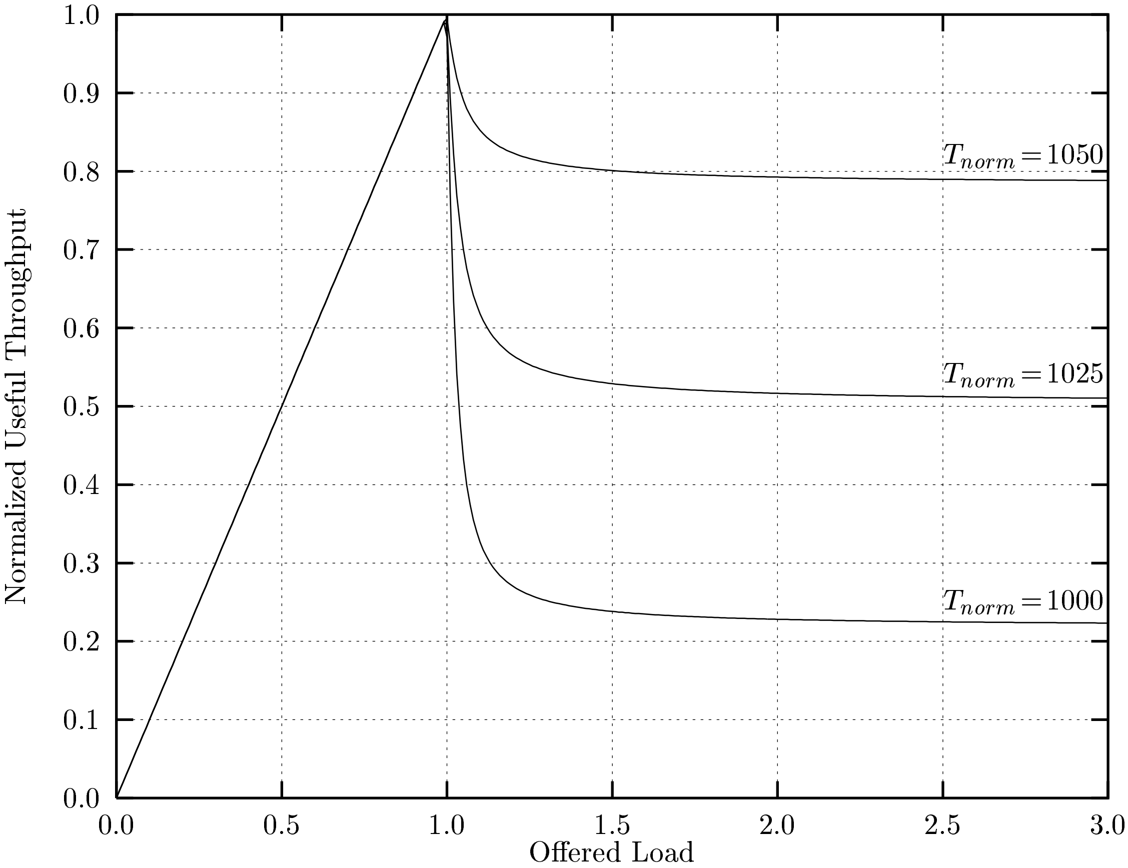

Figure 17: Normalized Useful Throughput vs. Offered Load with K=1025

There are some striking differences, however, between the analytical and simulation results. The asymptotic levels that the useful throughput approaches for $a \gg 1$ do not agree. The case of $T_{norm} = 1000$ corresponds to a timeout of 10 seconds for the simulated server whose $W_s = 0.01$ seconds. Whereas the analytical formula predicts a throughput of 0.22 for this case, the simulated result was around 0.40. Before explaining this discrepancy, let's gain some additional insight by discussing the analytical result for the case of $T_{norm} = 1025$.

The analytical asymptotic throughput for $T_{norm} = 1025$ is seen to be about 0.5. Since $K = 1025$ also, we have $K = T_{norm}$, and a 0.5 asymptotic value matches our intuition as we shall proceed to explain.

With $a \gg 1$, the server's queue will almost always be full (or nearly so). Assume that a request enters the queue, only to fill it up completely. This request must wait K service times before its service is complete. But what if the request's timeout is exactly K service times (i.e. $T_{norm} = K$)? Then the request will timeout with a probability close to 0.5. Why? Well, with K large, the sum of K service times is a random variable with a roughly

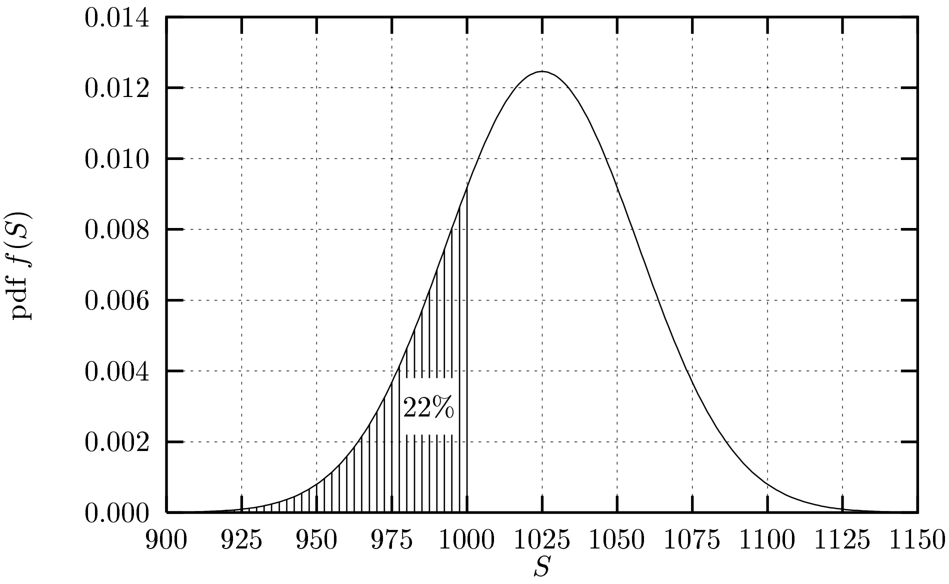

Figure 18: Density Function for S, the sum of 1025 exponential service times

normal distribution and with mean KW_s and standard deviation $\sqrt{K}\sigma_s$ (where W_s and σ_s are the single-service-time mean and standard deviation). Thus, the mean K-service-time wait is KW_s, or in normalized time units, just K. Since a normal random variable's density function is symmetric about its mean, our request with $T_{norm} = K$ is equally likely to have a wait shorter or longer than the mean K-service-time wait of K. Thus, it will timeout with probability ≈ 0.5. Since the server is effectively never idle (has a raw throughput of 1.0), we have for $a \gg 1$ and $T_{norm} = K$,

$$X_{useful}(a, K, T_{norm}) = raw\_throughput \cdot P[request\_timeout] \approx 1.0 \cdot 0.5 = 0.5$$

Now that we have more confidence in the analytical result, why then is there the discrepancy between the simulated and analytically-predicted high-load asymptotic throughput for $T_{norm} = 1000$? The answer lies with the variance of the service time, σ_s, as we shall now show.

The analytical model predicts a throughput of 0.22 for $a \gg 1$ for $T_{norm} = 1000$. Based on our discussion of the $T_{norm} = 1025$ case, we realize that a throughput of 0.22 means that approximately 22% of requests that enter the queue get serviced before they timeout. Said another way, the total wait for 1025 service times, S, is less than 1000 just 22% of the time. This is represented graphically in Figure 18 with the shaded region showing $P[S <= 1000] \approx 0.22$.

As mentioned before, the graph of Figure 18 approaches a normal curve for large K. The parameters of this normal are:

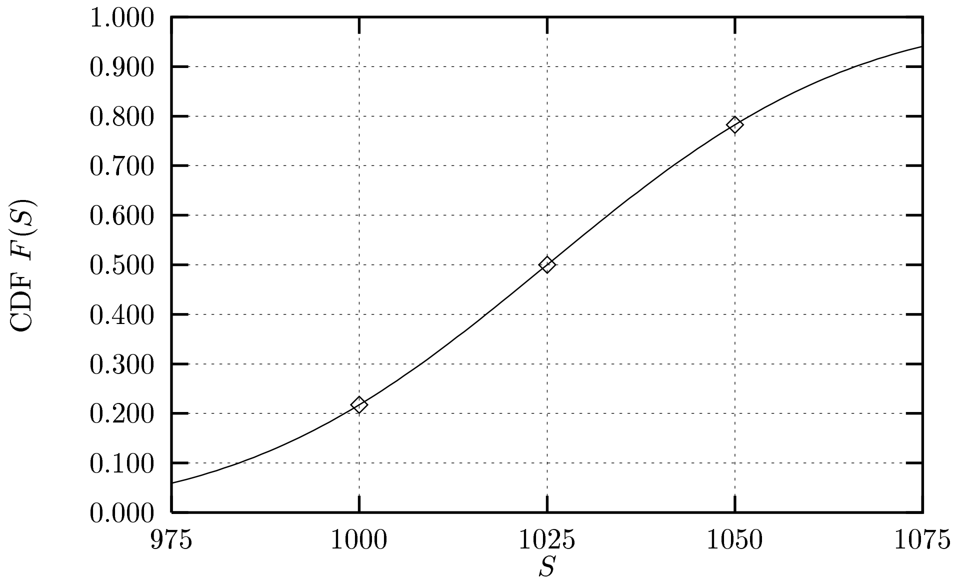

Figure 19: Distribution for S, the sum of 1025 service times with Coefficient of Variation = 1.0

	measured in secs.	normalized to W_s
mean	KW_s	K
std. dev.	$\sqrt{K}\sigma_s$	$\sqrt{K}\sigma_s/W_s$

where W_s and σ_s are the mean and standard deviation of a single service time. We can now see the importance of the σ_s/W_s value (the so-called coefficient of variation) for the service time. It is this parameter that governs the width of the normal density function for the sum of K service times, which in turn dictates the asymptotic levels of X_{useful} for offered loads $a \gg 1$.

This then explains why the theoretical and simulated results are different. For the analytical $M/M/1/K$ system, the exponentially-distributed service time has a coefficient of variation $\sigma_s/W_s = 1$, since $\sigma_s = W_s = 1/\mu$. However, the simulated workload used a SpecWeb-like service time distribution. This distribution was analyzed, as detailed in Appendix A.2, and found to have a coefficient of variation $\sigma_s/W_s = 0.037/0.01 = 3.7$. To see if these differences explain the discrepancy between the analytical and theoretical results, we can plot the cumulative distribution function for the sum of 1025 service times for $\sigma_s/W_s = 1.0$ and $\sigma_s/W_s = 3.7$.

Figure 19 shows this function for a coefficient of variation $= \sigma_s/W_s = 1.0$ and $K = 1025$. The values obtained for $T_{norm} = 1000, 1025$ and 1050 match closely the asymptotic levels of Figure 17 for those same T_{norm} values. These values, namely 0.22, 0.50 and 0.78, are marked with special points on Figure 19.

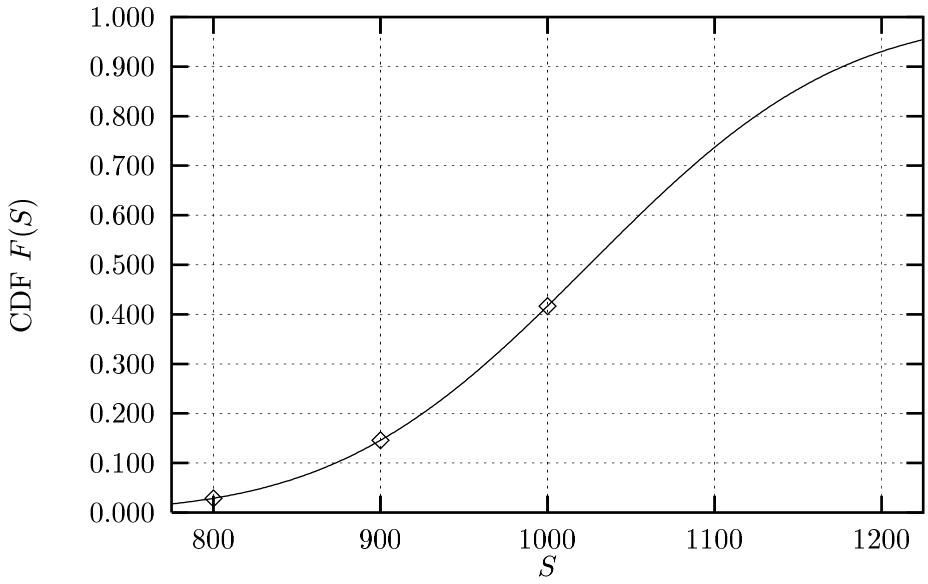

Figure 20: Distribution for S, the sum of 1025 service times with Coefficient of Variation = 3.7

More importantly, Figure 20, which plots the sum of 1025 service times having a coefficient of variation $= \sigma_s/W_s = 3.7$, shows strong correspondence with our simulated results in Figure 2. In particular, we can read off the values at the specially-marked points of $T_{norm} = 800, 900$ and 1000. These points, which correspond to the simulated timeouts of 8, 9 and 10 seconds respectively, show analytically-predicted asymptotic throughputs of .03, .15 and .42. These match the simulated asymptotic levels of Figure 2 shown on page 12!

Finally, we can express these results in terms of the standard normal distribution function $\Phi()$ as follows for $a \gg 1$:

$$X_{useful}(K, T_{norm}) = \Phi\left(\frac{T_{norm} - K}{\sqrt{K}\sigma_s/W_s}\right)$$

To summarize then, we have developed a simple formula to express a server's useful throughput as a function of the normalized offered load a, the system capacity K and the normalized client timeout T_{norm}. This formula, when plotted, was shown to agree qualitatively to the simulated results for the case of no DTOC and no client-retry represented in Figure 2. One discrepancy involved the asymptotic levels of throughput achieved for offered loads $a \gg 1$. We have shown that this discrepancy is explained by the different service times' coefficient of variation. A second formula was developed, based on the normal distribution, that accurately predicts the simulated asymptotic throughput levels.

6 Conclusion

In this paper, we proposed a method to detect timed-out client requests, the DTOC strategy, which can be used to significantly improve an overloaded server's performance. In support of this, we first described the problem of anxious and impatient clients that cancel and often resubmit their requests to servers. We further detailed how these canceled, i.e. timed-out, requests can cause loss of useful server throughput and, under some circumstances, a pathological system state we term request-timeout livelock. Using simulation models, we demonstrated the performance improvement of a server using the DTOC strategy and its capability for livelock avoidance, as compared to a server using a regular strategy. This we showed for a number of client request timeout values and for normal and busy-day workload scenarios. To minimize the overhead imposed by the DTOC strategy, we showed how it could be effective even if employed only in the presence of overload.

In addition, we described three possible implementations of the DTOC strategy for servers on a TCP/IP network. These implementations are based on different approaches for detecting a closed client-server connection and thereby inferring a timed-out client request.

Furthermore, we have presented some analytical methods to corroborate our simulation findings for the simple case of a server without the DTOC strategy and with no-retry clients. This study also showed the importance of the coefficient of variation of the single-request service time distribution for predicting the throughput for excessive loads.

We feel that DTOC is a promising technique for addressing the problems of servers under heavy load. Another such technique, session based admission control (SBAC) as introduced in [CP98], can be used in a complementary fashion to DTOC. In fact, DTOC can improve SBAC performance for workloads with short average session length, where for high traffic loads there is a large percentage of retried clients requests, as was shown in [CP98].

To minimize the overhead imposed by the DTOC strategy, we showed how it could be effective even if employed only in the presence of overload. However, many applications of DTOC may consider employing the strategy at all times, since detailed measurements of some production servers have shown that up to 25% of all requests are aborted, even during non-overloaded periods [P98]. Also, the low 0.5% overhead imposed by the DTOC strategy is negligible compared to the 20% to 470% percent performance gains experienced during overloaded periods.

Finally, there is room for future study into variations of the DTOC strategy. We showed that the useful server throughput is still not 100% under heavy loads for the DTOC implementation we chose to simulate- useful throughput was still lost to client requests that timed-out during their response preparation. Variant DTOC strategies that involve interrupt-driven notification of request timeouts or in-service polling for these timeouts are possibilities for future investigation.

7 Acknowledgements

The authors would like to thank Peter Phaal for very stimulating discussions on overloaded web server behavior.

Stephane Eranian deserves a special acknowledgement for his careful technical reviewing of the method and approach.

8 References

[A90] Arnold O. Allen: *Probability, Statistics, and Queueing Theory with Computer Science Applications. Second Edition.*, Academic Press, San Diego, CA, 1990.

[CI97] P.Cao, S.Irani: Cost Aware WWW Proxy Caching Algorithms. Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS), Monterey, CA, pp.193-206, December 1997.

[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-, June, 1998.

[HP-WebQoS] Press release: HP's WEBQoS allows businesses to deliver fast, consistent, dependable e-services. http://hpcc522.corp.hp.com:7380/retail/pr$_q$*os.htm*

[HTTP1.1] Hypertext Transfer Protocol – HTTP/1.1.
URL http://www.ietf.org/internet-drafts/draft-ietf-http-v11-spec-rev-05.txt

[M97] Manley, S. An analysis of Issues Facing World Wide Web Servers. Harvard University Tech Report, TR-97-12, May 1997.

[MCS97] Manley, S., Courage., M., Seltzer., M. A. Self-Scaling and Self-Configuring Benchmark for Web Servers. URL http://www.eecs.harvard.edu/~margo/papers/

[P98] Phaal, P. Private communication, May, 1998.

[Schwetman95] Schwetman, H. Object-oriented simulation mode ling 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/

[W96] Stephen Wolfram: *The Mathematica Book. Third Edition.*, Cambridge University Press, Cambridge, UK, 1996.

A APPENDIX

A.1 Simplification of the Analytical Throughput Equation

The 2 nested summations found in equation 8 from Section 5 made its graphing using *Mathematica* particularly slow. We repeat this equation below and develop a simpler form with 1 fewer summation:

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1-a^K-e^{-T_{norm}}\sum_{n=0}^{K-1}(1-a)a^n\sum_{i=0}^{n}\frac{T_{norm}^{\,i}}{i!}\right]$$

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1-a^K-(1-a)e^{-T_{norm}}\sum_{n=0}^{K-1}\sum_{i=0}^{n}a^n\frac{T_{norm}^{\,i}}{i!}\right] \qquad (10)$$

Note that we sum $a^n\frac{T_{norm}^{\,i}}{i!}$ over the combinations of i and n marked with the $\bullet$ below:

$$\begin{array}{cc|cccccc}
 & & & & i & & \\
 & & 0 & 1 & 2 & \cdots & K-1 \\
 & 0 & \bullet & & & & \\
 & 1 & \bullet & \bullet & & & \\
n & 2 & \bullet & \bullet & \bullet & & \\
 & \vdots & \bullet & \bullet & \bullet & \ddots & \\
 & K-1 & \bullet & \bullet & \bullet & \cdots & \bullet
\end{array}$$

The inner summation scans across the rows of the above diagram, whereas the outer summation advances the scanning from the top row down. However, we can reverse the order of the scanning, i.e. have the inner summation scan down the columns and have the outer summation advance the scanning from the left column towards the right. This transformation, when applied to equation 10, yields:

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1-a^K-(1-a)e^{-T_{norm}}\sum_{i=0}^{K-1}\sum_{n=i}^{K-1}a^n\frac{T_{norm}^{\,i}}{i!}\right]$$

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1-a^K-(1-a)e^{-T_{norm}}\sum_{i=0}^{K-1}\frac{T_{norm}^{\,i}}{i!}\sum_{n=i}^{K-1}a^n\right] \qquad (11)$$

Remembering that

$$\sum_{n=0}^{K-1}a^n = 1+a+a^2+\cdots+a^{K-1} = \frac{1-a^K}{1-a}$$

Then

$$\sum_{n=i}^{K-1}a^n = a^i\sum_{n=0}^{K-i-1}a^n = a^i\frac{1-a^{K-i}}{1-a} = \frac{a^i-a^K}{1-a}$$

Substituting this last result back into equation 11, we get

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1-a^K-(1-a)e^{-T_{norm}}\sum_{i=0}^{K-1}\frac{T_{norm}^{\,i}}{i!}\frac{a^i-a^K}{1-a}\right]$$

Canceling the $(1-a)$ term,

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1 - a^K - e^{-T_{norm}} \sum_{i=0}^{K-1} \frac{T_{norm}^{\;i}}{i!}(a^i - a^K)\right] \tag{12}$$

and behold, we have simplified our throughput equation down to just 1 summation. It is this form that we transfer back to section 5 as equation 9.

Now before we stop, it should be noted that some mathematics packages, such as *Mathematica* have an intrinsic and optimized support for numerous well-known functions, including the *incomplete gamma function*. It turns out that our throughput equation can be cast in terms of this function, thus eliminating (perhaps more accurately *hiding*) the last summation in our formula and improving further the speed of evaluation.

The upper incomplete gamma function is given as

$$\Gamma(k,T) = \int_T^{\infty} t^{k-1}\; e^{-t}\; dt$$

And if k is an integer K, we have the alternate form

$$\Gamma(K,T) = (K-1)!\, e^{-T} \sum_{i=0}^{K-1} \frac{T^i}{i!}$$

which can be restated as

$$\sum_{i=0}^{K-1} \frac{T^i}{i!} = \frac{e^T\, \Gamma(K,T)}{(K-1)!} \tag{13}$$

Repeating equation 12

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1 - a^K - e^{-T_{norm}} \sum_{i=0}^{K-1} \frac{T_{norm}^{\;i}}{i!}(a^i - a^K)\right]$$

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1 - a^K - e^{-T_{norm}} \sum_{i=0}^{K-1} \frac{(aT_{norm})^{\;i}}{i!} + a^K e^{-T_{norm}} \sum_{i=0}^{K-1} \frac{T_{norm}^{\;i}}{i!}\right]$$

Plugging in from equation 13,

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1 - a^K - e^{-T_{norm}} e^{aT_{norm}} \frac{\Gamma(K,aT_{norm})}{(K-1)!} + a^K \frac{\Gamma(K,T_{norm})}{(K-1)!}\right]$$

$$X_{useful}(a,K,T_{norm}) = \frac{a}{1-a^{K+1}}\left[1 - a^K - e^{(a-1)T_{norm}} \frac{\Gamma(K,aT_{norm})}{(K-1)!} + a^K \frac{\Gamma(K,T_{norm})}{(K-1)!}\right] \tag{14}$$

This then is our last, and in some sense most efficient, representation of the server throughput formula. It expresses X_{useful} only in terms of a, K, T_{norm} and the well-known incomplete gamma function $\Gamma(k,T)$.

	P[this_class]	File Size	P[this_size]		P[this_class]	File Size	P[this_size]
Class 0	0.35	100	0.0102	Class 2	0.14	10000	0.0041
		200	0.0167			20000	0.0067
		300	0.0276			30000	0.0110
		400	0.0455			40000	0.0182
		500	0.1500			50000	0.0600
		600	0.0455			60000	0.0182
		700	0.0276			70000	0.0110
		800	0.0167			80000	0.0067
		900	0.0102			90000	0.0041
Class 1	0.50	1000	0.0145	Class 3	0.01	100000	0.0003
		2000	0.0239			200000	0.0005
		3000	0.0394			300000	0.0008
		4000	0.0650			400000	0.0013
		5000	0.2143			500000	0.0043
		6000	0.0650			600000	0.0013
		7000	0.0394			700000	0.0008
		8000	0.0239			800000	0.0005
		9000	0.0145			900000	0.0003

Figure 21: Simulated Workload Filesize Distribution

A.2 Details of the Simulated Service-time Distribution

The service time distribution used in the simulated model is similar to the SpecWeb96 file-size distribution, and consists of the 4 file-size classes defined in Figure 21.

Note that we are implicitly assuming that the time required for a server to respond with a file of a given size is directly proportional to that file's size. This allows us to talk about response time in the somewhat unnatural unit of "bytes". Using "bytes", however, does allow us to better visualize the work that the server is performing and to feel comfortable with the various classes of responses, their frequency and size.

Within each class of file, SpecWeb96 defines a bell-shaped probability density function about the mean file size of each class. We too use a bell-shaped pdf, although not exactly identical to SpecWeb's. As a result, we have always represented our workload as "SpecWeb-like".

Other parameters of interest for this distribution are:

mean file size	14675 bytes
std. dev. file size	54351 bytes
scaled service time	0.0100 secs
scaled std. dev. service time	0.0370 secs
coefficient of variation	3.7