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, USA
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 product for HP9000 servers [HP-WebQoS].

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.

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 consider the DTOC technique as complementary to session based admission control, introduced in [CP98]. DTOC strategy is implemented as a part of WebQoS product for HP9000 servers [HP-WebQoS].

2 Methods for Detecting Timed-Out Requests

If a client experiences a long response delay after sending a request to a web server, a typical client behavior is: client clicks the browser "stop" button, followed by the browser "reload" button to resend the request. 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. 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.

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 uni-directional 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:

1. place the socket connection in NON-BLOCKING mode using `fcntl()` or `ioctl()`;
2. do a `recv()` call using a `flags` parameter value of `MSG_PEEK`;
3. 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

SpecWeb96 [SpecWeb96] is the industry standard benchmark 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. 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.

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.

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). 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

In order to demonstrate the performance impact of processing undetected timed-out requests in an overloaded web server, how it can lead to a request-timeout livelock, and, finally, the advantages and improved server performance under DTOC strategy, we built a simulation model using C++Sim [Schwetman95].

The basic structure of the model is outlined in Figure 1. 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.

We partition output (simulated) 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);

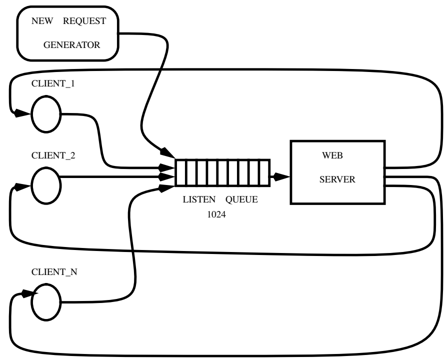

Figure 1: Basic Structure of the Simulation Model.

- *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 attempts, a "connection refused" message is often then presented to the user, who may then restart the whole procedure. We decided to simplify our simulation model by pronouncing the request *unsuccessful* upon the first failed attempt to connect to a server with a full listen queue.

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.

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.

4 Simulation Results

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 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

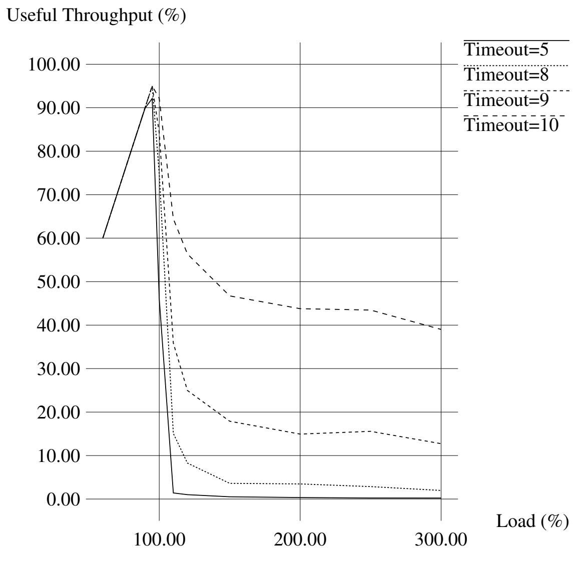

Figure 2: Throughput for a Server using the Regular Strategy and Clients with No Retry.

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 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.

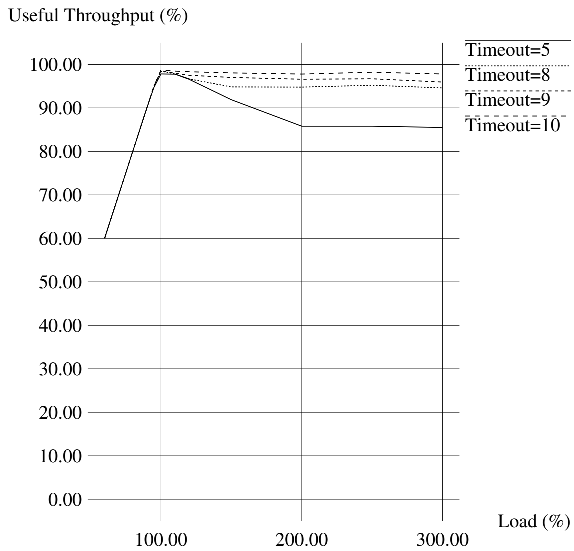

Figure 3: Throughput for a Server Augmented with the DTOC Strategy and Clients with No Retry.

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 $300\% \cdot 0.5\% = 1.5\%$. 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.

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.

The analysis presented above convincingly shows that a web server 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.

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 4.

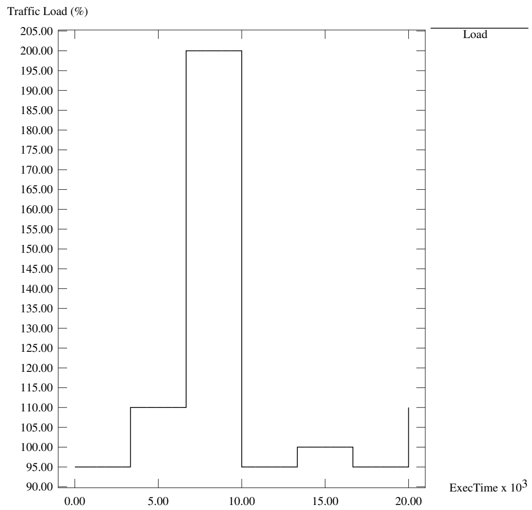

Figure 4: "Normal Day" Workload Traffic Pattern.

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.

The second workload is defined by the pattern showed in Figure 5. 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.

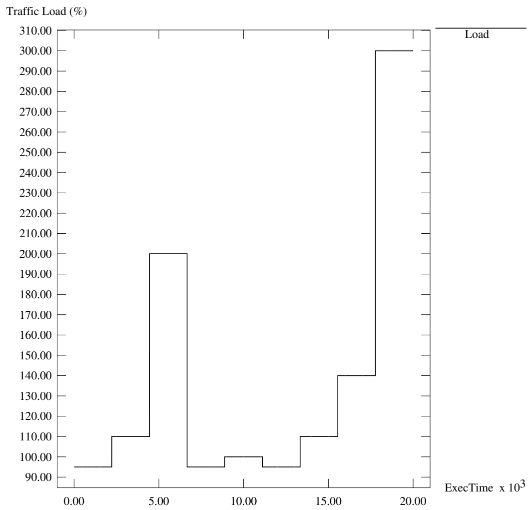

Figure 5: "Busy Day" Workload Traffic Pattern.

Figure 6 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.

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

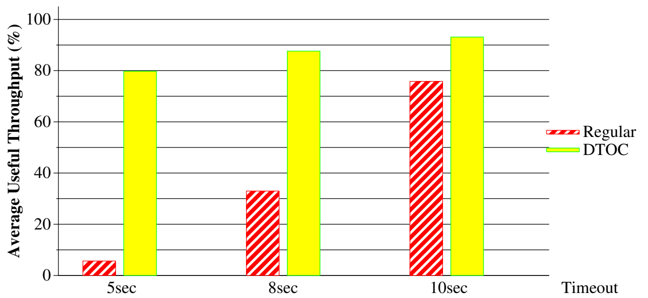

Figure 6: Throughput for a Server using the Regular Strategy against a Server Augmented with the DTOC Strategy with 1-Retry Clients for a "Normal Day" Workload.

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.

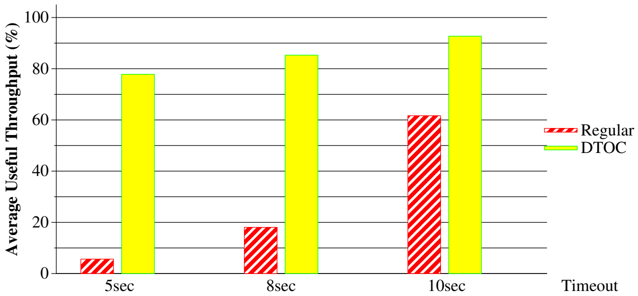

Figure 7: Throughput for a Server using the Regular Strategy against a Server Augmented with the DTOC Strategy with 1-Retry Clients for a "Busy Day" Workload.

Figure 7 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.

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 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.

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.

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.

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

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

[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. http://www.specbench.org/osg/web96/workload.html