

Minimizing the Impact of Orphan Requests in e-Commerce Services*

E. Kraemer G. Paixão D. Guedes[†] W. Meira Jr. V. Almeida

e-SPEED Lab — Dept. of Computer Science

Federal University of Minas Gerais

Belo Horizonte, MG 31270-010 Brazil

{kraemer,gopaixao,dorgival,meira,virgilio}@dcc.ufmg.br

Abstract

The most common problem of an overloaded electronic-commerce server is an increase in the response time perceived by customers, who may restart their requests hoping to get a faster response, or simply abort them, giving up on the store. Both behaviors generate “orphan” requests: although they were received by the server, they should not be answered because their requestors have already abandoned them. Orphan requests waste system resources, since the server becomes aware of their cancellation only when it tries to send a response and finds out that the connection was closed. In this paper we propose a new kernel service, the Connection Sentry, which keeps track of requests being performed and notify processes about an eventual cancellation. Once notified, a process can interrupt the execution of the request, saving system resources and bandwidth. We evaluated the gains by using our proposal in a virtual bookstore, where we observed that the Connection Sentry reduced service latency by up to 31% and increased the throughput by 27% in overloaded servers.

1 Introduction

Electronic commerce (e-commerce) servers often get overloaded with customer requests, which leads to noticeable increases in latency and to the degradation of the quality of the services provided. In both cases, customers may get frustrated by the slow process and give up on interacting with the store.

*This work was supported by CNPq-Brazil and Project SIAM-DCC-UFMG, grant MCT-FINEP-PRONEX number 76.97.1016.00.

[†]Sponsored by CNPq-Brazil grant number 300445/99-7

The interaction of users with an e-commerce server (a store) can be viewed as multiple user sessions, each of which starts when a user issues a first request and continues (through subsequent requests) until that user leaves, either with or without making a purchase. During each session users interact with the store by issuing multiple requests to locate desired products and get detailed information about them. Most of the pages returned by a server during a session are therefore dynamically created in response to the user’s specific needs.

As the number of clients accessing a virtual store increases, so does the server load, and that leads to longer response times. When that happens users tend to become impatient, taking one of two lines of action [4]: they may just give up their sessions and leave the site, or they may try to speed up the process by stopping the transfer and trying to restart it. The end effect may be the client leaving the site without a purchase.

Such restarting strategy has been studied in the context of traditional web servers [4], in which case it has been shown that restarting requests often is advantageous even if all customers employ it. However, in the context of e-commerce sites, the same argument is not valid. As we discuss in Section 2, the architecture of e-commerce sites is comparatively more complex than that of traditional web servers. Since most responses are dynamically created, each request may require a significant share of server resources to be fulfilled, and each reload in fact starts a new operation, increasing the server’s contention even further.

Current server architectures (which are built using the *sockets API* [8]) have no provision to notify a server when a client request becomes orphan in this way.

In this paper we present an extension to the sockets interface that allows a server to identify orphan sessions early and we explain how e-commerce servers can be adapted to make use of this extended interface to reduce their load. Section 3 describes the new interface and Section 4 shows some results of a virtual store using it. We discuss some related work in Section 5 and present some conclusions and future work in Section 6.

2 E-commerce sites

From the functional viewpoint, e-commerce sites are usually organized into layers that perform classes of services [7]. Typical services can be grouped into the following categories:

Presentation: the front-end of the electronic store. It is implemented by Web servers.

Business Logic: all application-related services. It is usually called the *application server*.

Database services: it provides persistent and reliable storage for the site's data.

The processing flow of a request in an e-commerce site structured in such a way is illustrated in Figure 1. A customer request arrives at the site as an HTTP request, which is then handled by the Web server. It may be a request for a static document (e.g., an html page or an image) or to execute an e-commerce function. In the latter case, the Web server invokes a function of the e-commerce server through a CGI-like script. The e-commerce server performs the requested service and most of the time needs to contact the database server.

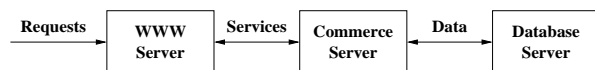


Figure 1: Flow of requests in an e-commerce site

It has been shown that in heavily loaded e-commerce sites the database server becomes the main bottleneck, followed by the application

server [7]. That is due to the large number of dynamic requests that require accesses to the store's persistent information.

In order to understand the problem of interrupted client sessions we must first understand how servers are implemented, how the information flows from one module of the store to the others, and how client behavior affects the operation of the servers.

```

/* For this, consider as a simplification */
/* that read and write calls do not return */
/* prematurely unless an error occurs. */
{
  int urlen, reslen, clisock, appserversock;
  char urlbuf[MAXURL], reponsebuf[MAXRESPONSE];

  /* Initialize sockets, accept new connection */
  urlen = read( clisock, urlbuf, MAXURL );

  /* In practice the url would be parsed here. */
  write( appserversock, urlbuf, urlen );

  /* Nothing to do until the reply arrives */
  reslen=read( appsvrsock,buf,MAXRESPONSE );

  if ( write( clisock, buf, reslen )!=reslen ) {
    /* Connection closed, client is gone */
  }
}
  
```

Figure 2: Simplified HTTP server code

2.1 Server implementation

All major HTTP server implementations are built around the *sockets API*. Figure 2 shows an extremely simplified version of the code at the heart of an HTTP server which handles every request received in an e-commerce site. That code illustrates the application-oriented nature of the interface: it is always the application who starts any interaction with the kernel, both to send and receive data. Only on those moments can the kernel notify the application of any connection-related event.

That, by the way, is in opposition to the protocol processing that takes place within the kernel: in that case, processing is event-triggered, occurring each time a message arrives or a timer expires. For example, data is received and processed by the kernel as soon as a network packet arrives. But that data will only be delivered to the application when it contacts the kernel to ask for it.

2.2 The dynamics of a session

Figure 3 shows the usual sequence of events triggered by a client request in a store organized in three levels. A significant amount of time may pass from the moment the HTTP server contacts the application server with the user's request until a reply is sent to the client with the information provided by the application server.

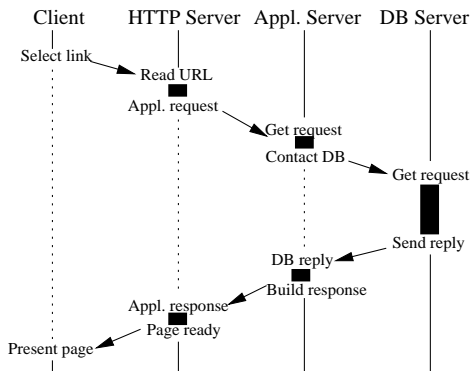


Figure 3: Complete request in current systems

Problems arise if the client gives up waiting for a response and leaves the site or stops the transfer and tries to restart loading the page. In both cases, if we consider standard HTTP 1.0 browsers, the end effect is that the current connection to the client is closed (and a new one is opened, in the case of a retry). Browser and server complete the handshake to close the connection and an EOF marker is put in the server's input stream by the kernel.

It is only when the server sends the first packet of the response to the client that a RESET packet is sent to the server indicating that the connection has been completely closed by the client. Only on the first attempt to write to the client after the RESET has been received will the server get an error signal indicating that the client is not listening anymore.

The problem is therefore illustrated in Figure 4: the application continues to compute the response to the client's request unaware of the fact that the client is not interested in that anymore. Unfortunately, the server will only become aware of that fact when it tries to access the connection again to write the response back.

That is true whether the server uses separate threads for each connection, as illustrated here, or

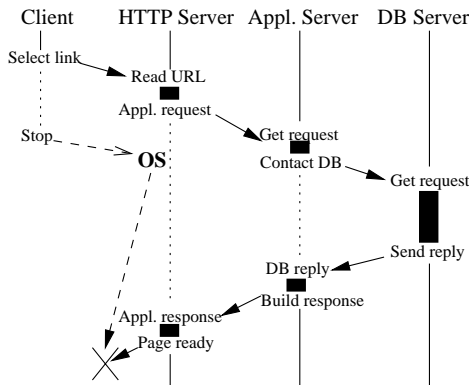


Figure 4: Orphan request in current systems

an organization based on a central demultiplexing around the `select()` system call.

In summary, each time the client leaves the site or hits the browser stop button prematurely, a request is left in the e-commerce server. That request consumes resources and increases the load, but will serve no useful purpose. In the next section we present a novel service that minimizes the problem of orphan requests.

3 The Connection Sentry

The new service described here is aimed at solving the problem of lack of a notification channel between kernel and application. The new interface should allow applications to register with the kernel when they want to be notified of such unexpected terminations, and allow the kernel to notify the application promptly if that happens. In addition, the application must be altered to take whatever actions it considers necessary to stop any processing related to the (now orphan) request.

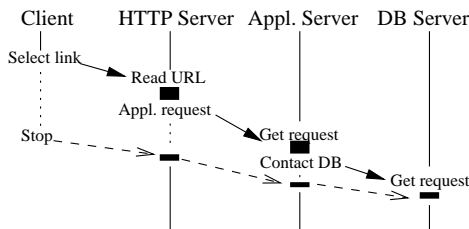


Figure 5: Orphan request with the Sentry

The intended effect is illustrated in Figure 5.

After the HTTP server starts working on a client's request, any activity in that connection (either a close or the arrival of new data) should be immediately notified to the application so that it can decide whether it should abort the processing triggered by the first request. That might require just a local abort, or it might also require notifying other servers involved.

3.1 Sentry implementation

Since a connection shutdown by the client is an asynchronous event in terms of the server's activity, we decided to use signals for the notification. The service was implemented as a new system call with which an application registers its interest in a given connection, identified by an open socket:

```
socket_watch( int sockfd, int signum );
```

When the kernel detects a shutdown in a connection which has been registered it interrupts the process with a signal.

Besides the usual error conditions (invalid parameters, etc.) the `socket_watch()` system call also returns an error if the connection associated with the socket has already been closed by its remote endpoint, and that can be identified by checking the value of `errno`. That was made to cover cases where the shutdown handshake may happen even before the server application has had the time to process the request and register with the kernel.

The implementation required adding new fields to the process descriptor and to the TCP control block to keep the information necessary to link processes to the connections they are interested in and *vice-versa*. When a process registers with the kernel, its identifier is added to a list of *interested processes* in the TCP control block. Every time a connection receives a FIN packet and starts the shutdown handshake the kernel verifies if there are processes registered for that connection. If so, the requested signal is sent to each of them.

3.2 Changes to the servers

In order to use the new kernel service the server must be altered to include code that identifies those client requests which will require extra processing.

If such a request is received, the server process registers the signal handler and the socket to be watched by the kernel. Only after that it proceeds to compute the response.

If the client does not close the connection the computation ends, the server unregisters the socket and proceeds to send the response to the client. On the other hand, if the client closes the connection before the server is finished the signal handler is started by the kernel to cleanly terminate any processing in the server for that request and to notify other processes started on its behalf.

It is the responsibility of the application programmer to provide the handler and to make sure that it can abort the computation once it receives the proper signal. Halting an ongoing transaction cleanly is highly dependent on the semantics of the application, so it cannot be done automatically.

4 Case study: e-bookstore

In order to validate the new kernel service we applied it to a virtual bookstore that we developed in the e-SPEED lab. The sections that follow describe how the site architecture was adapted to handle the information about requests that should be discarded, the workload used in the experiments and the results obtained.

4.1 Architecture of the store

Our store follows the basic architecture described in Section 2. It uses Apache as the HTTP server, MySQL as the database server and an optimized, locally developed store [7]. The store provides six services¹: *i) home, ii) search, iii) browse, iv) select, v) add to cart, vi) buy*. In addition to these, the store also answers to *file* requests for information related to the products. The original store was adapted to make use of the new service as illustrated in Figure 6.

The Apache server was altered to determine first whether the application server and the database server have to be contacted to build the requested page. If that is the case, it registers the

¹In the results we omitted *add to cart* and *buy* because just a few of them were present in our workload and the measurements were not statistically significant.

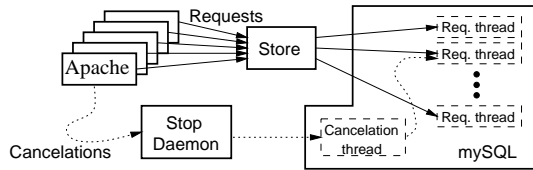


Figure 6: Organization used for the case study

socket for that connection. Each Apache process that receives a registered signal notifies a *stop daemon* responsible for concentrating the control of orphan requests.

To achieve this, the stop daemon keeps a permanent connection to the database server (located in a separate machine). When it receives the *url* from an Apache server, it just forwards that information to the database server over that permanent connection.

As originally implemented, requests arriving at the *mySQL* database server are queued as new threads which wait on a lock to access the database. The server was altered to create a separate thread that listens to notifications from the stop daemon. Upon receipt of a cancellation, that thread marks the thread associated with the request as canceled. If that thread has not gained access to the database yet it will check that mark after acquiring the lock but before it starts the actual computation and it will terminate without processing the request.

4.2 Workload/experimental setup

To verify the gains provided by the new service we ran two sets of comparative experiments on the virtual bookstore, each comparing the performance of a store on a standard Unix system to that of the a store using the Sentry. In the figures and tables that follow, the two runs are identified by “X” (Unix) and “S” (Sentry-enabled), respectively.

All experiments use a modified version of Surge [3] that handles sessions, i.e., sequences of requests issued by a customer. The workload used is built from an actual trace from a large electronic bookstore comprising 123,664 requests distributed over 3,177 user sessions (38.92 requests per session on average) and our system had the product database of the same store, comprising 4,309 books. Each experiment run lasted 30 minutes and the re-

sults presented in the sections that follow show the averages obtained from three separate experiment runs. The variance of the measurements among all runs are within 7%.

In every run client requests are divided into two sets: control and test. Clients are Surge processes replaying a part of the original trace, so they do not map directly to customers, but to groups of customer sessions. The client processes in the control set request services and objects without cancellations, while the clients in the test set cancel requests following a certain patience function.

In the results shown we consider the measurements from the control set only, since we want to quantify the impact of the Sentry on the customers that continue to use the store while other customers (the test set) cancel some requests. We model impatience through a mathematical function that describes the probability of the customer canceling a request after a given number of seconds.

Each server runs on a different 200 MHz Pentium with 64 MB RAM. The Web server is an Apache 1.3.9 server running on top of FreeBSD 3.1. All other machines, including the Surge clients, execute Linux 2.2.12. The database server runs our modified version of *MySQL* 3.22.22. All machines are on the same LAN segment.

4.3 Impact of the patience level

| Class of request | Patience level | | | | | |
|------------------|----------------|------|------|------|------|------|
| | 7.5 s | | 10 s | | 15 s | |
| | X | S | X | S | X | S |
| home | 2.6 | 0.3 | 1.6 | 0.3 | 1.0 | 0.4 |
| browse | 4.5 | 3.5 | 8.2 | 3.7 | 7.8 | 3.8 |
| search | 15.2 | 12.0 | 13.9 | 12.8 | 13.6 | 13.2 |
| file | 1.2 | 0.1 | 0.7 | 0.1 | 0.4 | 0.1 |
| select | 8.2 | 5.7 | 6.7 | 5.9 | 6.5 | 6.1 |

X: standard Unix interface; S: using Connection Sentry

Table 1: Average latency for various patience levels (in seconds per request)

In this section we evaluate the impact of varying the patience level on the efficacy of the Connection Sentry. We show results for three patience levels, with maximum customer waiting times of 7.5,

10, and 15 seconds².

As the patience level drops, users are more likely to leave the store if a request is not served promptly. The results can be observed in Tables 1 and 2. As expected, the gains in terms of latency decrease as the patience level increases, since the customers wait more before canceling a request. Considering search requests, we got an improvement of over 20% for the 7.5-second patience function (average latency dropped from 15.2 seconds to 12.0 seconds) compared to almost no improvement when customers more wait 15 seconds or more.

| Class of request | Patience level | | | | | |
|------------------|----------------|------|------|------|------|------|
| | 7.5 s | | 10 s | | 15 s | |
| | X | S | X | S | X | S |
| home | 353 | 621 | 441 | 608 | 500 | 600 |
| browse | 30 | 33 | 33 | 33 | 33 | 33 |
| search | 345 | 441 | 383 | 414 | 395 | 409 |
| file | 1029 | 1312 | 1173 | 1289 | 1216 | 1278 |
| select | 61 | 69 | 66 | 69 | 66 | 69 |

X: standard Unix interface; S: using Connection Sentry

Table 2: Average throughput for various patience levels (in number of requests)

Similar results can be observed in terms of throughput. For instance, by using the Connection Sentry, the servers were able to increase the number of *home* requests by 76% (from 353 to 621) and *search* requests answered by 28% (from 345 to 441) for a patience level of 7.5 seconds.

4.4 Impact of the client workload

In order to evaluate the impact of the client workload we varied the number of simultaneous client processes issuing requests to the e-commerce server. Again, client processes were divided into control and test sets. We performed experiments using 3, 6, and 9 client processes per set. The patience level in these experiments was 7.5 seconds. Again, we use just the measurements from the control set.

The results can be seen in Tables 3 and 4. The latency reduction for *search* requests ranged from 21% for nine client processes in each set to 6% when

²Results for patience levels of 6 and 30 seconds are almost indistinguishable from 7.5 and 15 seconds, respectively.

| Class of request | Number of client processes | | | | | |
|------------------|----------------------------|------|------|-----|-----|-----|
| | 9 | | 6 | | 3 | |
| | X | S | X | S | X | S |
| home | 2.6 | 0.3 | 0.2 | 0.2 | 0.2 | 0.2 |
| browse | 4.5 | 3.5 | 2.4 | 1.3 | 0.4 | 0.4 |
| search | 15.2 | 12.0 | 10.4 | 8.5 | 3.9 | 3.7 |
| file | 1.2 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| select | 8.2 | 5.7 | 4.6 | 3.7 | 1.7 | 2.1 |

X: standard Unix interface; S: using Connection Sentry

Table 3: Average latency for various workloads (in seconds per request)

just three client processes were active. As it would be expected, there is almost no gain for the lighter workload, for which orphan requests do not represent a problem in terms of wasting resources. For throughput, again, the best gains are in the heavier workload, where the number of home requests served increased by 76% and search requests by 28%.

| Class of request | Number of client processes | | | | | |
|------------------|----------------------------|------|-----|------|-----|-----|
| | 9 | | 6 | | 3 | |
| | X | S | X | S | X | S |
| home | 353 | 621 | 355 | 375 | 225 | 226 |
| browse | 30 | 33 | 30 | 30 | 25 | 25 |
| search | 345 | 441 | 395 | 472 | 402 | 408 |
| file | 1029 | 1312 | 949 | 1119 | 660 | 673 |
| select | 61 | 69 | 59 | 60 | 38 | 38 |

X: standard Unix interface; S: using Connection Sentry

Table 4: Average throughput for various workloads (in number of requests)

By checking the profile of the transaction server (Figure 7), we can observe that the use of the Connection Sentry reduces significantly the multiprogramming level on that server. In particular, the elapsed times associated with building pages are reduced by almost 45%. We observe similar reductions in the costs for the other tasks, which are not visible in the graph due to their reduced overall significance in the application server.

5 Related work

There has been a lot of work on improving the performance of traditional web servers. Mogul [5]

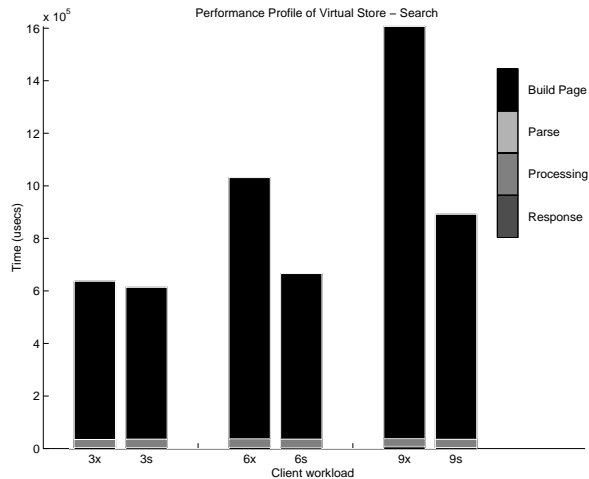


Figure 7: Transaction server profiles

pointed out the problems in using traditional system architectures for heavily loaded servers. Banga and Mogul [1] have shown how the current *sockets* interface can be improved to better handle the large number of concurrent connections in a loaded server. Others, like Nahum *et al.* [6] have proposed extensions to the operating system interface with new system calls intended to combine multiple operations previously performed by separate primitives, reducing system call overheads.

A more extensive reorganization of the operating system kernel and interfaces has also been proposed by Banga *et al.* [2]. That work describes a new system call that makes it easier for the application to handle events associated with network connections. It might be interesting to study how the two ideas could be combined together.

6 Conclusions

In this paper we presented the Connection Sentry, a novel approach for minimizing the impact of orphan requests in multi-layer Internet Servers. We implemented and evaluated it by using a virtual bookstore on a trace-based workload. Results show that latency experienced by customers decreased by up to 31 % and bandwidth increased by up to 27%. Furthermore, we observed improvements regardless the customer “level of impatience” and server workload in terms of simultaneous sessions being served.

We intend to continue this work by making the application path transparent for different machines,

allowing notifications to remote servers to be handled. We are also investigating the use of a cleaner interface to the notification mechanism.

Acknowledgments

We gratefully acknowledge Thiago Maia for the first implementation of the service, which was our starting point for this work.

References

- [1] G. Banga and J. C. Mogul. Scalable kernel performance for internet servers under realistic loads. In *Proceedings of the 1998 USENIX Annual Technical Conference*, June 1998.
- [2] G. Banga, J. C. Mogul, and P. Druschel. A scalable and explicit event delivery mechanism for unix. In *Proc. of the USENIX 1999 Annual Technical Conference*, June 1999.
- [3] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *Proc. 1998 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, June 1998.
- [4] S. Maurer and B. Huberman. Restart strategies and internet congestion. Xerox PARC Technical Report, 1999.
- [5] J. C. Mogul. Operating systems support for busy internet services. In *Proc. of the Fifth HoTOS Workshop*, May 1995.
- [6] Erich Nahum, Tsipora Barzilai, and Dilip Kandlur. Performance issues in www servers. In *Proceedings of the ACM SIGMETRICS '99 Conference*, June 1999.
- [7] G. Paixão *et al.* Design and implementation of a tool for measuring the performance of complex e-commerce sites. In *Proceedings of the 11th Performance Tools Conference*, March 2000.
- [8] W. Stevens. *TCP/IP Illustrated, The Protocols*, volume I. Addison-Wesley, 1994.