[4] Proceedings of IASTED Conference on Applied Informatics, February 2003.

Enhance Features and Performance of a Linux-based Content Switch

C. Edward Chow and Chandra Prakash
Department of Computer Science,
University of Colorado at Colorado Springs,
Colorado Springs, CO 80933-7150, USA
{chow, cprakash}@cs.uccs.edu
Tel: (719)262-3110
FAX: (719)262-3369

Abstract

In this paper we discuss the problems encountered in the development of a Linux LVS-based content switch and present their solutions. A pre-allocate server scheme is proposed to improve the TCP delayed binding bottleneck, and performance of its implementation is presented. The content switch rule syntax is extended to allow the extraction of specific tag values in the XML requests.

Keywords: Content Switch, Cluster, TCP Delayed Binding, Load Balancing, Network Architecture

1. Introduction

The tremendous growth in World Wide Web usage has become a double-edged sword for operators of large Web Sites. On the one hand, increases in request volume translate into increased subscription, advertising, and hosting revenue. On the other hand, scaling web sites to meet this increased demand has become more and more difficult as the number of requests for content exceed a particular server's ability to respond. In the best case, users will experience degraded service, in the worst case the server can be driven to collapse resulting in a complete loss of service.

One approach to alleviate handling of large volume of requests is to distribute their load among a group of nearly identical servers [1]. A master controller, that can be a dedicated host or a process, first receives the requests and delegates it to the appropriate real server [2,3]. This describes a typical load balancing system. A content switch is such a front end of a load balancing system that distributes load based on the content of the received requests.

There are conventional ways of load balancing at the transport layer, i.e., Layer 4 of TCP/IP. One of them is to use the port number of the incoming request and direct it to a real server responsible for handling the response for that specific port. For example, if the port number in the incoming request is 21 it can be routed to machine catering FTP requests and if the port number is 80, it is routed to host running HTTP server. This mechanism cannot differentiate among requests with different content.

The web based content switch uses the content of web request to select a real server [4,5,6,7]. For example, a content switch can make routing decisions based on URL of incoming web request. In electronic commerce systems, a content switch may route the incoming request based on the purchase amount, or the customer ID contained in the XML content of the request. The routing decisions are typically expressed in terms of rules where conditions classify the packets for different routing actions.

The terms in these rule conditions can include a matching function for checking whether the URL matches a regular expression. For example, the rule "*if (match(URL, ".gif$") { routeTo(imageServer)}*" allows the routing decision based on the file extension of the request. The terms can also be a relational operation on a XML tag value. For example, rule "*if (xml.purchase/totalAmount>50000) {routeTo(highSpeedServer)}*" allows the purchase of higher amount to be treated differently. This requires the efficient parsing of the XML document in the payload for specific XML tags that are referenced in content switching rule set. We have implemented a Linux LVS-based content switch, LCS version 0.1, uses Network Address Translation (NAT) based IP virtual service [8,9]. It allows the specification of such rule syntax.

Since many network services are based on TCP protocol, the web switch needs to perform three-way handshake (Sync, Sync-Ack, Ack message exchange) with the client before it can receive the application level content. After selecting the real server based on the content, the content switch needs to go through three-way handshake with the chosen real server and then relay the application layer request. This is called *TCP delayed binding*. Since the sequence numbers committed by the content switch and that by the real server for the session are different, the content switch needs to convert the sequence numbers in every packet that follows. This introduces a lot of packet processing overhead.

In this paper, we present a pre-allocate server scheme that can reduce the TCP delay binding processing overhead. We have also discussed the problems encountered in the development of the LCS and presented their solutions. Based on our experience and problems encountered during the design and implementation of LCS, we present suggestions for better component design and overall improvement of content switching systems.

2. Problems and Solutions for Content Switch Design

In this section we discuss the content switch design issues related to content processing, improving TCP delayed binding, and client request buffering.

2.1. Handling request with multiple packets

If the client's request is too big to fit in one TCP segment, the content switch has to wait for all the segments that comprise that request before commencing the rule matching. This is especially true of non-idempotent HTTP requests like PUT and POST, and for e-commerce applications with large XML request. This further gives rise to the following sub-problems that we had to account for:

Determine the content length

We had to determine the content length of the variable incoming data stream in order to flag the end of client request. The content length information of such request can be obtained from the "Content-Length" meta-header in the HTTP request. However, the value of the content length itself can span across multiple segments as shown in the example below:

TCP Segment n contains:

POST /cgi-bin/cs622/purchase.pl HTTP/1.0\r\n
Referer: http://archie.uccs.edu/~acsd/lcs/xmldemo.html\r\n
Connection: Keep-Alive\r\n
User-Agent: Mozilla/4.75 [en] (X11; U; Linux 2.2.16-22enterprise i686) \r\n
Host: viva.uccs.edu\r\n
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*\r\n
Accept-Encoding: gzip\r\n
Accept-Language: en\r\n
Accept-Charset: iso-8859-1,*,utf-8\r\n
Content-type: application/x-www-form-urlencoded\r\n
Content-length: 7

TCP Segment n+1 contains:

53\r\n
data (753 bytes)

As seen in the above example the individual bytes of the content length are split across two consecutive TCP segments, the first segment contains 7 and the next segment contains the remaining two byte, i.e., 53. This is true for any field within the HTTP request header, even for the sequence of data bytes that form the "Content-Length" string.

Fragmentation of application level content

After the content length is determined, the content switch can then wait for all the packets of the same request. Typically, these packets are saved in different memory area. In Linux, they are saved in sk_buff structures linked by double link list. Each of these data structures contains the timestamp, TCP/IP headers, followed by the content payload. Therefore, the actual content is fragmented and spread out in these network buffers. Extracting URL field in the HTTP request is easy, since it is in the first packet. But for extracting other meta-headers and especially the XML tag values in the content field of the HTTP request, the fragmentation of the TCP payload content post difficult challenging problem for the content switch designers. One approach is to concatenate all individual non-contiguous TCP segments back to back into one coherent buffer, that can then be used for XML parsing, or pattern matching. Another approach is to redesign the XML parsing or pattern matching so that they can work with data that spread across several segments. A specialized memory address mapping hardware similar to the translation look-aside cache used in virtual memory system can also help speed up the packet processing.

The first approach requires the expensive memory copying and uses additional memory. The original TCP segments are not released after the concatenation of their payload content, since once the real server is selected, these TCP segments will be modified and sent to the chosen real server. The modification includes the destination IP address field, possibly the TCP port field, the ACK sequence number, and very importantly the checksum.

While buffering client data, the content switch has to send ACK's for the segments that comprise the client request, otherwise the client TCP will assume that the server is dead or is very slow, and will not send subsequent packets. This is achieved by invoking appropriate ACK sending routines from the IP layer of the content switch.

For large sized (> 40K) client requests, we also observed some of the relayed segments were dropped by the chosen real sever. Further analysis indicated that the problem is due to the segment relay by the content switch is implemented in IP instead of TCP layer. The data sending was done continuously from the queued buffers without considering the window advertised by the TCP stack of the real server. This flooding of data caused the real server to drop some of the received TCP packets. It was observed that the acknowledgment number sent by the real server was held constant, even though the content switch had emptied all buffered data. The result was that there was no response seen from the real server, as it had not acknowledged receipt of all data. This problem was solved by having the content switch keep track of the real server acknowledgment number along with buffering of last packet sent to it. When the acknowledgment sent by the real server was less than the next sequence number of the packet to be sent subsequently, the last sent packet was retransmitted. The next sequence number is computed from the sequence number of the last packet and size of data in it. This retransmission helped alleviate packet flooding at the real server and ensure all client data are properly received.

2.2. Handle Different Data Encoded Methods

There are two basic ways for submitting the XML-based request to the web server. One is to use the form with text input or text area input. The other is to use submit it as XML document. When submitting it with form, the XML request data are encoded using the x-www-form-urlencoding method and the "Content-Type" meta-header will have the value of

"x-www-form-urlencoded". When submitting it as XML document, the "Content-Type" meta-header will have the value of "text/xml" and the content are submitted in plain text without further encoding. With the latter encoding type, all special characters like line feed (\n), carriage return (\r), left anchor (<) and right anchor (>) etc. retain their ASCII representation. In the former encoding type the special characters have encodings like "%XX", where XX is the hexadecimal representation of ASCII value of that special character. For example, for the "x-www-form-urlencoded" encoding type, the values for the exemplified special characters will be "%0A", "%0D", "%3C" and "%3E" respectively. Hence, the rule matching module need to correctly parse the XML content of the client request depending on the content type.

2.3. Allow Referencing Specific XML Tags

The rule specification scheme should be flexible enough to account for the exact tag name or rule field indicated in the rule specification. Here is an example that illustrates this point. Consider the XML document:

```
<purchase>
  <customerName>CCL</customerName>
  <customerID>111222333</customerID>
  <item>
    <productID>309121544</productID>
    <unitPrice>5000</unitPrice>
    <subTotal>50000</subTotal>
  </item>
  <item>
    <productID>309121538</productID>
    <unitPrice>200</unitPrice>
    <subTotal>2000</subTotal>
  </item>
  <totalAmount>52000</totalAmount>
</purchase>
<purchase>
<customerName>CDL</customerName>
  <customerID>111222444</customerID>
  <item>
    <productID>30913555</productID>
    <unitPrice>3000</unitPrice>
    <subTotal>20000</subTotal>
  </item>
<totalAmount>20000</totalAmount>
 </purchase>
```

In the above XML document, some of the tags are repeated, e.g., purchase, item, totalAmount. Hence a rule syntax is needed to allow for selecting a particular set of tags in the rule set. Here is an example of a scheme that addresses this problem. To specify a rule based on subTotal value present in the second item tag within the first purchase tag, the condition of the rule will be specified as "purchase:1.item:2.subtotal > 5000". As another example, "purchase:2.totalAmount < 15000" specifies the condition of a rule based on the totalAmount tag present within the second purchase tag.

2.4. Handle Long Transactions in SSL and Email network services

In our Linux-based Content Switch, the content/header extraction and rule matching are performed in the kernel to avoid unnecessary copying. However, we have found that for network services that requires long computation and interface with other packages, some of the packet processing functions are better handled at the application level. For example, there are a lot of packages, including McAfee's uvscan and AMAVis scanmail, mutt (recombine email component), for detecting and removing email virus, but almost all of them are implemented in application level and interact with the sendmail program. It will require significant effort to rewrite them as kernel modules. Same observations were derived on SSL processing.

SMTP goes through long message exchange between the client and the server. The client sends a sequence of messages including HELO, MAIL FROM, RCPT TO, Data, followed by the actual body of the message. The server will respond with the specific code and confirmation message. Therefore the important email addresses for the sender and the receiver will appear at different stages of the transaction. The content switch needs to be able to store these messages in the buffer. Once the related header information is extracted and rules matched, these messages will be forwarded to the real mail server. For spam mail removal, the sending email address is extracted from the MAIL FROM message. For incoming email load balancing, the receiving email address is extracted from the RCPT TO message. Compared with SMTP, the processing of IMAP or POP is much simpler, since we only need to wait for the login in USER message for load balancing rule matching, but they have the same requirement for storing and forwarding the message sequence to the real server.

3. Design of Pre-allocate Server Scheme

The content switch has to buffer application level data for rule matching before the selected real server can be chosen. This is called TCP *Delayed Binding* as the response is delivered to the client after some delay associated with the buffering of client data and the rule matching [2]. This is shown in the Figure 1. Here we assume that the size of the http request can fit in one TCP payload and the request are sent in one IP packet. In real networks, we have observed that a request got split into two IP packets even though there are less than 800 bytes. In Figure 1, it is also assumed that the return document is small and can be fit into one IP packet. In typical web access, the return document is sent over multiple IP packets. CSEQ is the sequence number chosen by the client. DSEQ is the sequence number chosen by the content switch on

behalf of the real server. SSEQ is the sequence number chosen by the real server. Note that at Step 9, the content switch needs to change the sequence number from the real server to DSEQ+1; while at Step 10, the ACK number needs to be changed to SSEQ. The modification of the sequence number and the IP address fields need be performed for all the following packets over the same session.

We implemented a heuristic solution to this TCP delayed binding problem, where the client and real server mapping is pre-allocated and stored in a hash table with (hash) key as the client address. When a client sends a request to the content switch for the first time, there would not be any entry in the hash table and the request will go via normal data buffering and rule matching scheme, and an entry will be added to the pre-allocate hash table with the client IP address as key and the real server address as data. When the same client happens to send the next request, an entry will be found in the pre-allocate hash table and the client request will be directly forwarded to the real server addressed by the matching hash table entry. This avoids the rule matching overhead.

Figure 2a shows the modified delay binding in the pre-allocate scheme, when the pre-allocate server is the right one. Figure 2b shows the message exchange among the pre-allocate server, the right server, the content switch, and the client, when the guess it wrong. Note that when the guess it right, the web access can be complete in six steps instead of ten steps, and there is no need for sequence number modification for Step5 and Step6. When the direct routing or IP tunnel scheme is used instead of NAT, the return document can be sent directly to the client and reduce the processing overhead at the content switch.

For subsequent requests, when there is a matching hash table entry found in the pre-allocate hash table, it may happen the real server specified by the matching entry may not be the correct real server for that request. In that case the pre-allocate scheme degenerates to the default case, where rule matching is done for the client request. The worst case scenario in the pre-allocate scheme is where the real server specified in the matching hash table entry comes out to be wrong. Hence, it mandates that the client data be always buffered as done in the default scheme.

The content switch must examine the response from the real server specified in the matching hash table entry, before applying the degenerate rule matching. If the response does not contain HTTP response code 200 (HTTP OK), then the content switch switches to the default scheme. If the response code is 200, we free up the queued client request.

In our implementation if the real server specified in the matching hash table entry and the real server selected via rule matching after a wrong pre-allocate guess are same, we allow the response from the wrongly guessed real server to be forwarded to the client.

4. Performance of Pre-allocate Server Scheme

To evaluate the performance of the pre-allocate server scheme, a testbed with one content switch and two real server was set up with the following configurations:

Machine Spec	IP Address	OS	Web Server
viva.uccs.edu P5 240MHz 128MB (Content Switch)	128.198.192.192	Redhat 6.2 running LCS0.2 kernel based on Linux 2.2-16-3	Apache 1.3.14
ace.uccs.edu P5 166MHz 64MB (Real Server 1)	128.198.192.198	Redhat 6.2 running LCS0.2 kernel based on Linux 2.2-16-3	Apache 1.3.14
vinci.uccs.edu P5 240 MHz 128MB (Real Server 2)	128.198.192.183	Redhat 6.2 running LCS0.2 kernel based on Linux 2.2-16-3	Apache 1.3.14

We compared the response times of various document size between basic TCP delayed binding scheme and the pre-allocate scheme with the following set of series as shown in Figure 3:

Series 1 - Basic scheme with no rule matching module inserted, i.e., using default IPVS.

Series 2 - Basic scheme with the rule matching module inserted.

Series 3 - Pre-allocate scheme with all hits, i.e., where all pre-allocate guesses were correct.

Series 4 - Pre-allocate scheme with all misses, i.e., where all pre-allocate guesses were wrong.

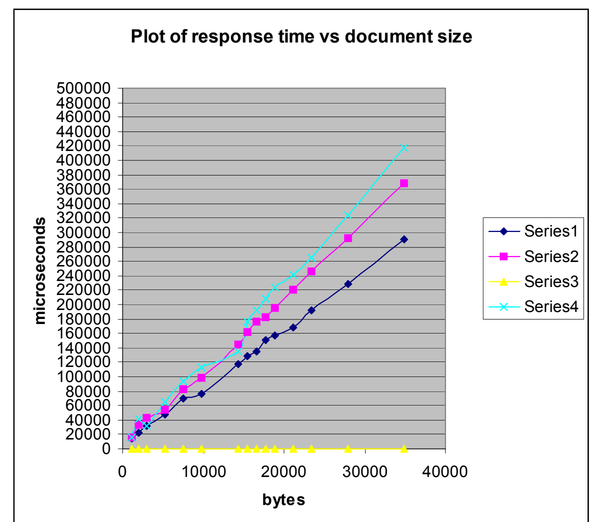

Figure 3. Performance of Pre-allocate Server Scheme

The response time represents the time difference between the time when the first packet for the request was seen at CS and the time when first packet of response of "correct" real server seen at CS. The document size represents the size of different set of HTTP POST requests used. As shown in the

Figure 3 the pre-allocate server scheme with all hits has almost constant response time whereas with all misses the response time is somewhat poorer than basic non-pre-allocate schemes. This is due to extra processing done in hash table lookup and delay associated with examining the response from the initial wrongly guessed real server. If we take the average case scenario to be somewhere between pre-allocate best case and pre-allocate worst case, we can see the improved performance in pre-allocate scheme.

The comparison between series 1 and series 2 obviously shows the overhead of rule matching.

The present version of content switch does not handle multiple requests in a Keep-Alive connection. It passes only the first request in a connection for rule matching. All subsequent requests, following the first request, in a given connection are routed to the (same) real server chosen via rule matching from the first request. Hence, a "true" content switch must be able to schedule all HTTP requests in a given connection via rule matching and must treat all requests uniformly. This will add processing overhead, as all requests in a given connection need to be examined.

In addition to the pre-allocate scheme we have implemented two other schemes and compare there performance. These two schemes are based on the premise that virtual server need not be the sole decision making component in a content switching system. If we distribute tasks between all working components in the content switching system, we can have better resource allocation and hence better throughput. This is manifested in the schemes described below.

The first scheme transfers the sequence number translation of the server response at the real server's end, instead of the virtual server. The data are queued, and delayed binding and the rule matching are the same as in the basic scheme, but once the real server is selected, it will be sent information about the sequence number expected by the client TCP. The real server in this scheme will directly send its response to client without virtual server being the default gateway. Hence, in this case the real server is aware of the virtual server unlike the basic scheme. A new layer were implemented between TCP and IP stack of each real server, which will translate the sequence number of response data as that expected by the client in forward path. In the reverse path, i.e., for ACKs sent by client for the response data, this intermediate layer will serve to translate the ACK sequence number sent by client to ACK sequence number expected by the real server. As evident in this approach, the virtual server is freed up from having to deal with translating server and client sequence numbers.

The second scheme, called filtering, transfers both rule matching and sequence number translation to the real server. The client request will be multicast to all real servers. Until the appropriate real server is chosen, the virtual server TCP will send ACK to client like the basic scheme, but there will be no data buffering at the virtual server. Each real server will respond with a response code indicating its alacrity for the client request. The real server may also send some load balancing information to assist real server selection at the virtual server. The virtual server will use the response code from each of the real server to issue a final "voting" decision as to who will serve the client request. The selected real server will be allowed to send its response and information about client expected sequence number will be sent to it. A TCP reset will be sent to the rejected real servers. The real server will send its response to client directly in this case too.

5. Conclusion

We have discussed the problems encountered in the design and implementation of Linux LVS-based Content Switch and presented our solutions to these problems. A pre-allocate server scheme for improving the TCP delayed binding performance is proposed and implemented. The performance results of the content switch with the basic TCP delayed binding and that of pre-allocate server scheme are presented. It shows that the pre-allocate server scheme improves the average performance of a web based content switch.

6. References

1. High Performance Cluster Computing: Architectures and Systems, Vol. 1&2, by Rajkumar Buyya (Editor), May 21, 1999, Prentice Hall.
2. George Apostolopoulos, David Aubespin, Vinod Peris, Prashant Pradhan, Debanjan Saha, “ Design, Implementation and Performance of a Content-Based Switch**”,** Proc. Infocom2000, Tel Aviv, March 26 - 30, 2000, http://www.ieee-infocom.org/2000/papers/440.ps
3. Linux Virtual Server (LVS) documentation, http://linuxvirtualserver.org/Documents.html.
4. “Foundry ServIron Installation and Configuration Guide,” May 2000. http://www.foundrynetworks.com/techdocs/SI/index.htm.
5. “Intel IXA API SDK 4.0 for Intel PA 100,”http://www.intel.com/design/network/products/software/ixapi.htm and http://www.intel.com/design/ixa/whitepapers/ixa.htm#IXA_SDK.
6. F5 BIG IP, http://www.f5.com/f5products/bigip/bigipwhitepapers.html.
7. CISCO Content Services Switch Configuration guide, http://www.cisco.com/univercd/cc/td/doc/product/webscale/css/css_410/advcfggd/index.htm.
8. C. Edward Chow and Weihong Wang, “Design and Implementation of a Linux-based Content switch,” to be published in Proceeding of International Conference on Parallel and Distributed Computing, Applications, andTechniques (PDCAT) 2001, Taipei, Taiwan.
9. C. Edward Chow Ganesh Godavari, and Jianhua Xie, “Content Switch Rules and their Conflict Detection,” to be published in Proceeding of International Conference on Parallel and Distributed Computing, Applications, andTechniques (PDCAT) 2001, Taipei, Taiwan.

[4] Proceedings of IASTED Conference on Applied Informatics, February 2003.

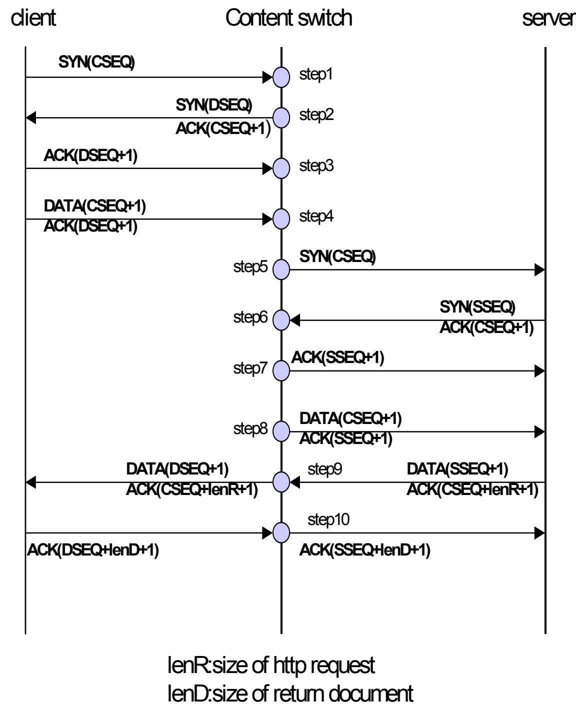

Figure 1. TCP Delayed Binding.

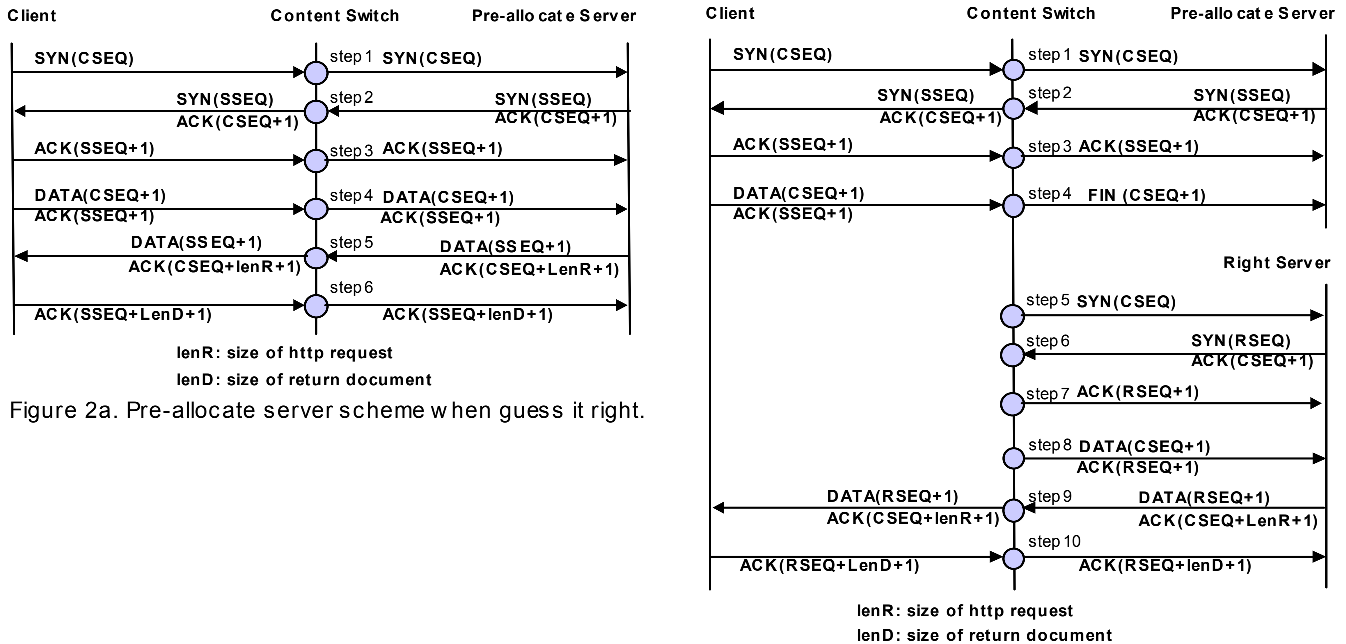

Figure 2a. Pre-allocate server scheme when guess it right.

Figure 2b. Pre-allocate server scheme when guess it wrong.