CHAPTER 4

Layer 7 Load Balancing and Content Customization

This chapter will discuss the methods and protocols involved in accomplishing a Layer 7 load-balancing solution. The reasons for and benefits of deploying persistent and customized load balancing will be explored.

TCP is the building block for Layer 7 protocols, such as the HTTP. This chapter will examine the interaction between HTTP and TCP and how client and server communication at the TCP and HTTP layers are accomplished.

Following the discussion of the TCP and HTTP, we will present a case study involving the requirement for Layer 7 load balancing from the application, security, and infrastructure perspective.

Benefits of Layer 7 Load Balancing

Load balancing for most applications can be accomplished using the basic information about the clients and the services that they are trying to reach, be that web-based content or VPN concentrators. Usually these decisions are made at the IP and TCP layers, which include looking at either the source or destination IP address or the destination TCP or UDP port number. However, as the applications and services offered become more complex, there is an increasing need to provide load-balancing decisions at layers above the transport layer (the TCP layer). This is where Layer 7 load balancing comes into the picture. Layer 7 load balancing provides inspection into the packet payload and identification of headers and fields to allow for more intelligent load balancing of user requests. The decision could be based on various HTTP method URLs or HTTP protocol headers, which we will discuss in detail in the section "Introduction to HTTP."

The three major reasons for Layer 7 load balancing are:

- Scalability and acceleration of the application
- Persistent user sessions on a server
- Content customization based on user profile

The following sections describe these advantages.

Scalability and Application Acceleration

As the number of clients of popular web-based applications increase multifold over a year, scaling internal and external applications is one of the key worries of data center application teams. Typically, the applications are scaled by adding more servers with replicated content and adding real servers to the Layer 4 load balancer. This works well for the networking team but is a cumbersome activity for the application team. This is because any change in the application or server content needs to be updated on more servers.

Layer 7 load balancing provides a solution which is desirable by both the network and application teams. The load balancer with its hardware-based Layer 7 packet inspection capabilities can be used to direct clients to different groups of servers. The server load balancing (SLB) devices can look into the URL and distinguish between various content requests. For example, content distribution can be performed as users make request for an URL, such as http://www.cisco.com/**partner**/index.html, which can be distinguished from an URL http://www.cisco.com/**cust**/index.html for the same domain. An SLB device can direct users for partner and cust to separate server farms respectively. Thus any change to partner-related content would only need to be updated on the partner server farm. Application distribution can also be performed by distinguishing */*.cgi from */*.html.

This is not just a method but also a great tool for improving server and application management, as it makes a substantial difference in the end user's experience. Because of dedicated content and application servers that will function faster, users will notice faster page download times.

Session Persistence

User session persistence to applications is another key benefit of Layer 7 load balancing. As the user's request is load balanced to a particular server, it needs to be persisted to that server. This is critical, as a client's authentication or shopping cart (browsing history) may exist in one server and not in the others. If the client's next TCP connection is load balanced to another server that does not have the client's history or session information, then the client would have to start from scratch.

In typical load-balanced environments, session persistence is provided by using the source IP sticky method. The source IP sticky method is used by the load balancer to track client connections to the application based on the client's source IP address. Any time the client with the same source IP address makes a connection via the load balancer, the load balancer will stick or forward that client's connection to the same application to which the client initially connected. This works well, but it can result in uneven load balancing when a large number of clients visit the site from behind mega proxies. Since mega proxies NAT multiple client source addresses to a single IP address, multiple clients can be using the same IP address; and if the source IP sticky method is enabled on the load balancer, multiple client connections will be forwarded to the same application, resulting in uneven load balancing.

Layer 7 load balancing session persistence can be based on an HTTP cookie, a URL, or a SSL session ID. This enables the load balancers to distinguish between users and also provide persistence for connections to servers even when they have the same source IP address.

Content Customization

As the global world is adapting to the Internet rapidly, providing customized content based on language or geographic region is becoming increasingly important. By inspecting the HTTP header requests, content can be inspected at the Layer 7 level (the HTTP protocol level) and connection can be redirected to a geographically or linguistically appropriate server or site. For example, if a client is making connections in Chinese, its requests should be catered to the servers that serve content in Chinese.

Introduction to TCP

The TCP was designed to provide reliable mechanisms for communications between a client and a server. Since TCP ensures the integrity of the data being transferred, not only in the proper sequence but also in utilizing the network bandwidth optimally, it is applied as a reliable connection-oriented protocol. A good understanding of TCP is essential for Layer 7 load balancing. This is the protocol used by HTTP and SSL. In the following sections, we will cover the details of the TCP protocol, which is a standard-based protocol defined in RFC 793, including:

- Data segments
- TCP headers
- TCP connection establishment and termination
- TCP flow control

Data Segments

In order to provide reliability for data communication, TCP provides retransmission and sequencing of the data segments being carried between the client and the server. A message sent from the client to the server or vice versa is called a *segment* in the TCP world. In order to ensure that a segment has reached the destination, the receiver sends an *acknowledgement* to the sender.

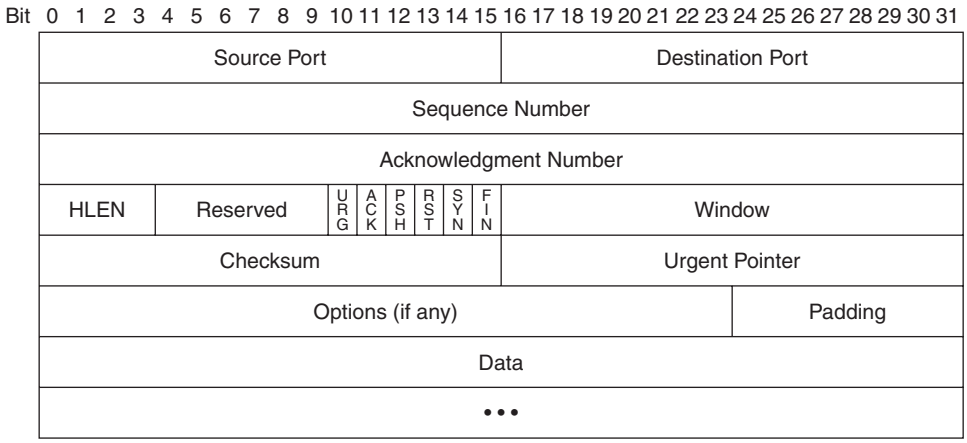
The postal service is a great analogy that can be used to understand TCP/IP. For example, when one sends a certified package in the mail, a message for the receipt of the package is sent to the sender as soon as the package arrives at the destination. Similarly, once the data has been successfully received in its original form, the receiver sends an acknowledgement back to the sender that it has received the data properly and the data is in its original and undamaged state. If the sent data is damaged or changed, the receiver simply ignores it. After a certain specified time, if an acknowledgement from the receiver has not been received, the sender checks to see the status of the sent data. Using TCP as the Layer 4 transport protocol ensures reliable communication and data integrity between the client and the server.

TCP Headers

In this section, we will cover some of the important TCP header fields, such as source and destination port, sequence number, acknowledgement number, header length, and more.

Figure 4-1 illustrates the structure of the TCP header. The size of the TCP header is 20 bytes when the Options fields are not set.

Figure 4-1 *TCP Protocol Header*



The following subsections provide descriptions of the TCP header fields.

Source and Destination Port

A TCP segment is carried within an IP packet; thus the source and destination IP addresses of the client and server are present within the IP header. The TCP header starts with a 16-bit Source Port field and a 16-bit Destination Port field. This port is the communication channel that the client and server use to transmit TCP segments to and receive TCP segments from each other. A very common example for identifying these values is when a client makes a connection to a web server on TCP port 80 (the TCP Destination Port), while the source port for a client is usually a random port above the well-known reserved ports of 1024. As many clients with random source ports make connections to a server on port 80, the way for the server to distinguish between the connections as being separate from client to client is to identify the connection based on the source IP, destination IP, source port, and destination port. Since the destination IP and port stay the same, the IP address and source port combination has to be unique in order for a connection to be unique.

Sequence Number

The 32-bit Sequence Number field in the TCP header indicates the first byte of data (the start of the data byte) in this particular segment. This is important in order to track and accomplish the reassembly of the TCP segments on the receiving end.

Acknowledgement Number

The 32-bit Acknowledgement Number field is used to identify the sequence number that the sender of the acknowledgement is expecting to receive.

Header Length

The 4-bit Header Length field is the total length of the header. It is needed because some of the OPTIONS bit fields can be of variable length. The maximum size for the Header Length fields when all the possible OPTIONS are selected is 60 bytes; otherwise, it is a 20-byte header.

Reserved

This 6-bit field is reserved and will be used for future enhancements to the TCP header.

Control Bits

The control bits are six flag bits, used in identifying and handling the TCP segment. The following are the control bits used:

- *URG*—The Urgent Pointer (URG) flag indicates to the receiver to accept data as it is deemed urgent.
- *ACK*—The Acknowledgment (ACK) flag identifies the successful reception of the segment and the next data byte, as specified in the Acknowledgment Number field, to be expected by the sender from the receiver.
- *PSH*—The Push (PSH) flag, when set, tells the receiver to immediately forward or push the data to the application.
- *RST*—The Reset (RST) flag is used to abort an existing connection and reset it so that the buffers holding the data can be cleared.
- *SYN*—The Synchronization (SYN) flag is used to signal that the sequence number between the receiver and the client need to be synchronized. The SYN bit is used during the initial connection setup between the client and the server.
- *FIN*—The Finished (FIN) flag is used to tell the receiver that the sender is finished sending the data and the receiver can close its half of the connection.

Window

The 16-bit Window field is for flow control using an advertised window size, which is the amount of bytes the sender and receiver are willing to accept during the exchange of TCP segments. The maximum size for the window is 65,535 bytes.

Checksum

The 16-bit *checksum* value is used to verify the integrity of the TCP headers and the data. If a segment contains an odd number of header and text octets to be checked, the last octet is padded on the right with zeros to form a 16-bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.

The checksum also covers a 96-bit pseudo header conceptually prefixed to the TCP header. This pseudo header contains the source address, the destination address, the protocol, and TCP length. This gives the TCP protection against misrouted segments. This information is carried in the IP and is transferred across the TCP/IP interface in the arguments or results of calls by the TCP on the IP.

The checksum value is stored in the Checksum field, and the pseudo header is discarded. On the receiving end, a similar operation is performed and the values are checked. If the checksum does not match, the segment is discarded.

Urgent Pointer

The 16-bit Urgent Pointer field indicates to the receiver that this value must be added to the Sequence Number field to produce the last byte of the urgent data (primarily when the urgent data ends). This field is used in conjunction with the URG flag.

Options

For added features and functionality, TCP has reserved the OPTIONS field. Depending on the option(s) used, the length of this field will vary in size, but it cannot be larger than 40 bytes due to the size of the header length field (4 bits). The Maximum Segment Size (MSS) option is the most common one used. The MSS is used to negotiate between the sender and the receiver the maximum size of the segment they will transfer. Other options for flow and congestion control such as time stamp for TCP segments can also be set using the OPTIONS field. The following are option codes that can be used in the OPTIONS field:

- No-Operation

  This option code may be used between options, such as to align the beginning of a subsequent option on a word boundary. There is no guarantee that senders will use this option, so receivers must be prepared to process options even if they do not begin on a word boundary.

- MSS

  If this option is present, then it communicates the maximum receive segment size at the TCP that sends this segment. This field must only be sent in the initial connection request (that is, in segments with the SYN control bit set). If this option is not used, any segment size is allowed.

Padding

The TCP header padding is used to ensure that the TCP header ends and data begins on a 32-bit boundary. The padding is composed of zeros.

In the next section, we will discuss the TCP header fields in a lot more detail.

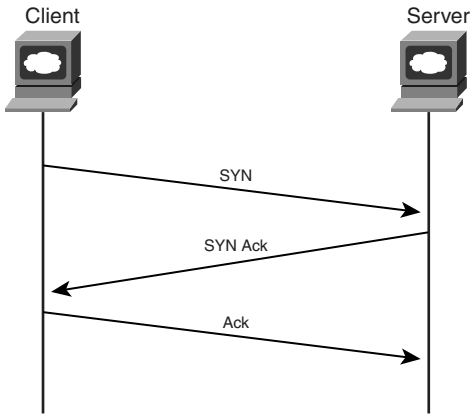
TCP Connection Establishment and Termination

There are various steps that need to be completed before data is transferred to and from a client and server. The steps include the negotiations between the client and the server, which are part of the TCP connection establishment phase. Similarly, when the data transfer is complete, the client and server go through the steps of tearing down the connection so that other processes on the client and server can use them. These steps are part of TCP connection termination phase.

TCP Connection Establishment

TCP relies on the connection initiation and setup based on control information called a handshake. This handshake is an exchange of control information between the client and the server before data can be transferred. The handshake or the connection setup uses a three-step process called the three-way handshake. If we follow the communication between host A and host B, the TCP three-way handshake can be illustrated as shown in Figure 4-2.

Figure 4-2 *TCP Connection Establishment*



As shown in the figure, the following are steps that specify the details of the TCP connection establishment phase:

1. A client initiates the connection by sending a TCP segment with the SYN flag set. This segment tells the server that the client wants to establish a connection. The segment also contains the sequence number that the client will use as a starting number for its

segments. The segments are used to synchronize the data, as they might arrive out of order on the client or server receiving the data.

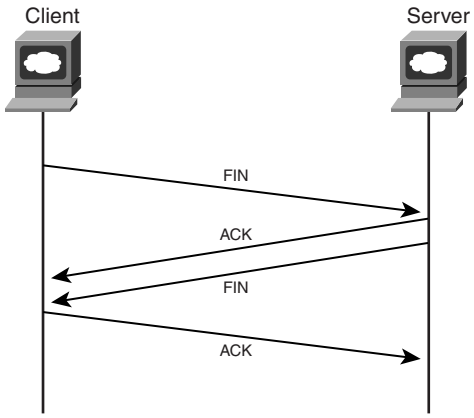
2. In response to the SYN segment received from the client, the server responds to the client with its own segment, which has the SYN bit and the ACK bit set (basically acknowledging the segment sent from client to server). The server will also set the sequence number that it will use for the communication.

3. Finally, the client sends a segment that acknowledges receipt of the server's segment, and this is the start of the transferring of data.

TCP Connection Termination

When the server and the client finish with the data transfers, they will conduct another exchange of segments containing the FIN bit set to close the connection. As opposed to the connection establishment phase, the connection termination phase includes the exchange of four segments. Because in TCP, the segments are arriving independently of each other in each direction, each connection end must shut down independently of the other. The connection termination phase can be illustrated as in Figure 4-3.

Figure 4-3 *TCP Connection Teardown*



As shown in the figure, the following are steps that specify the details of the TCP connection termination phase:

1. The client generates a FIN segment because the application running on the client is closing the connection.

2. The server receives the FIN and signals its application that the client has requested to close the connection. The server immediately acknowledges (ACK) the FIN from the client.

3 As the application on the server decides to shut down, it initiates a FIN segment to the client to close the connection.

4 Upon receipt of the FIN segment from the server, the client acknowledges it with an ACK.

TCP Flow Control

The TCP flow control techniques are implemented to optimize the transfer of data between the client and the server along with the network parameters. The following sections discuss these flow control techniques:

- TCP acknowledgements, retransmission, and timeout
- Sliding window

TCP Acknowledgements, Retransmission, and Timeout

TCP manages the data it sends as a continuous byte stream, and it has to maintain the sequence in which bytes are sent and received. The Sequence Number and Acknowledgement Number fields in the TCP header keep track of the bytes.

To ensure that the segments of the data stream are properly received, both the client and the server need to know the segment's initial sequence number. The two ends of the connection synchronize byte-numbering systems exchanging SYN segments during the handshake. The Sequence Number field in the SYN segment contains the initial sequence number (ISN), which is the starting point for the byte-numbering system.

Each byte of data is numbered sequentially from the ISN, so the first real byte of data sent has a sequence number of ISN + 1. The sequence number in the header of a data segment identifies the sequential position in the data stream of the first data byte in the segment.

Since TCP's underlying network protocol is IP, one of the challenges that TCP faces is to manage segments that are received out of order or are lost. TCP has to manage all flow control and retransmissions because IP does not have any mechanisms to do so. During the TCP data transfer phase, the receiver acknowledges the longest contiguous prefix of stream that was received in order. Each segment is acknowledged, providing continuous feedback to the sender as the data transfer progresses. The TCP acknowledgement specifies the sequence number of the next octet that the receiver expects to receive. For example, if the first byte sent was numbered 1 and 5200 bytes have been successfully received, the acknowledgement number will be 5201.

For identifying the successful receipt of a segment, each time a TCP segment is delivered, there is a timer that starts and waits for an acknowledgement. If the timer expires before the segment has been acknowledged by the receiver, the TCP sender assumes that the segment

was lost or corrupted and needs to be retransmitted. This timer is referred to as the retransmission timeout (RTO). This mechanism adapts to the changes in the networks and delays in acknowledgments and adjusts the RTO.

Sliding Window

TCP uses the Window field to manage the number of bytes the remote end is able to accept. If the receiver is capable of accepting 1000 more bytes, the window would be 1000. The window indicates to the sender that it can continue sending segments as long as the total number of bytes that it sends is smaller than the window of bytes the receiver can accept. The receiver controls the flow of bytes from the sender by changing the size of the window. A zero window tells the sender to cease transmission until it receives a non-zero window value. Both the sender and the receiver advertise the number of bytes each is willing to receive from the other. The window size reflects how much buffer size the receiving end is capable of handling. The sender has to obey the size of the window before delivering the segment.

Now that we have reviewed TCP and understood the session establishment and teardown, we will discuss our core Layer 7 protocol, HTTP, which rides on TCP.

Introduction to HTTP

When the Internet was first designed, the designers had to come up with the solution of clients being able to retrieve resources from servers. One of the most common features of a web page is a hyperlink. This is the clickable link on the web page that points to other resources.

For this concept to work, the uniqueness of the documents and names had to be globally maintained. The naming convention on the Web that is used to maintain the uniqueness of resources (web pages) is accomplished by the URL. The other issue that web designers had to address was how these resources would be represented and formatted in a uniform and readable format. This problem was solved by HTML. Web designers defined HTTP to determine how the various formats, such as text, graphics that make up the web content (HTML), and web page names (URLs), are transported from the client to the server and back.

The HTTP is the most common protocol for transferring resources on the Web. HTTP defines the format and meaning of messages exchanged between web components, such as clients and servers. A protocol is simply a language, similar to natural languages used by humans, except that it is used by machines or software components. In the next sections, we will look at the following:

- Protocol details
- HTTP header field
- Differences between versions

The protocol definitions can be found in RFC 2068, which defines HTTP version 1.1.

Protocol Details

HTTP is a stateless protocol. Statelessness implies the absence of a state maintenance, during the client and server communication. The HTTP protocol does not have any awareness of the previous client or server request or response. The decision not to maintain state in the HTTP protocol was to provide scalability on the Internet, where a large number of clients could be making connections to the server. If the server started to maintain state for each connection, the resources and the time for connections would increase drastically, hampering end user experience. However, for applications that did require state across the multiple HTTP requests, other enhancements and headers were included (such as cookies) to satisfy the requirements. Figure 4-4 shows a client browser accessing www.example.com.

Figure 4-4 *A Client Browser Accessing www.example.com*

Following are some of the key details of the HTTP header.

HTTP Methods

A request method notifies the HTTP server of what action should be performed on the resource identified by the requested Uniform Resource Identifier (URI). The request method is included in a client's request along with several headers and a URI. The method is applied to the resource by the origin server, and a response is generated. The response consists of a response code, metadata information about the resource, and the other response headers. Following are some of the key HTTP request methods.

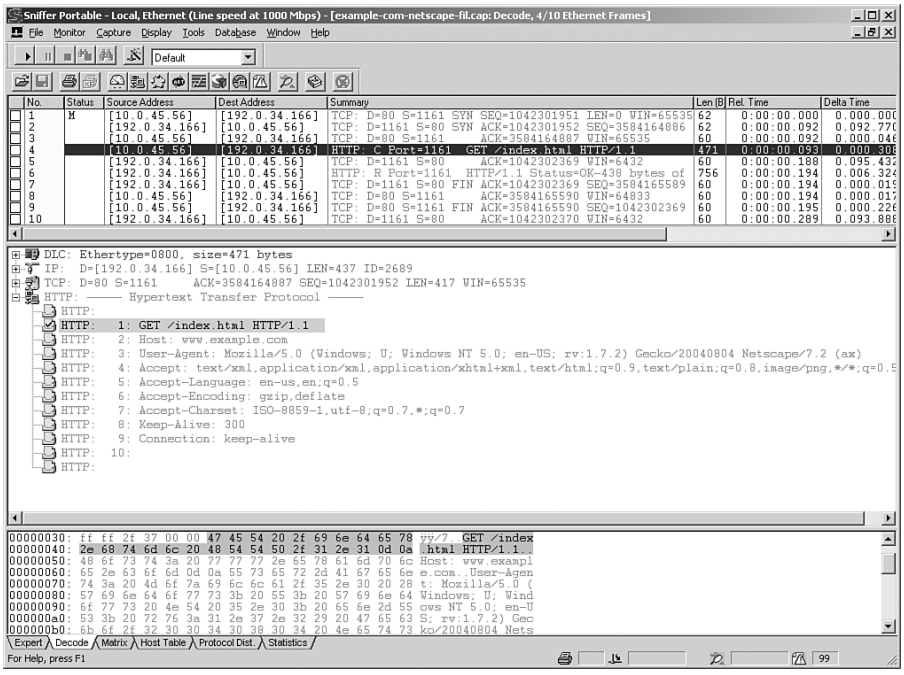
GET Method

The GET method requests a document from a specific location on the server. This is the main method used for document retrieval. The response to a GET request is returned to the requesting client. If the URI refers to a static file, a GET request will read the file and return it to the client. If the URI refers to an executable program or a script, the result of the program or script is returned as part of the body within the entity body portion of the request.

The GET method can be constructed to add modifier headers to yield different results; for example, if the If-Modified-Since modifier is used with the GET request along with a specified date, the server sends the appropriate response code based on the changes made to the resource according to the date specified.

Figure 4-5 shows a Sniffer capture of an HTTP GET request.

Figure 4-5 *Sniffer Capture of an HTTP Get Request*



HEAD Method

The HEAD method is used to obtain the metadata information for the resource. There is no response body returned as a result of a HEAD request. However, the metadata that a server returns should be the same metadata that would be returned if the request method had been GET. One of the biggest advantages of using the HEAD method is to check the status of the resource without the overhead of the resource being returned. This method is widely used in SLBs to provide probes to check the resource availability of a server before being brought into rotation for load balancing client requests. Another advantage is that the HEAD method uses the modification time of a document.

POST Method

The POST method allows clients to provide data input to a data handling program, such as an executable script running on the server. The server on which this data handling program is being executed allows only for specific actions to be performed. Since the POST method can potentially change the contents of the resource, the clients need to have access rights to execute the process on the server. The POST method can be used for Common Gateway Interface (CGI) programs, gateway-to-network services, CLI programs, and database operations. In a POST request, the data sent to the server is in the entity body of the client's request. After the server processes the POST request and headers, it may pass the entity body to another program for processing. In some cases, a server's custom application programming interface (API) may handle the data, instead of a program external to the server. POST requests should be accompanied by a content-type header, describing the format of the client's entity body. The most commonly used format with POST is the URL-encoding scheme used for CGI applications. It allows form data to be translated into a list of variables and values.

PUT Method

The PUT method is similar to the POST in that processing in the method would typically result in a different version of the resource identified by the URI in the request. If the request URI does not exist, it is created, and if it already exists, it is modified. However, the resource identified in the PUT method alone would change as a result of the request. When using the PUT method, the data is sent as part of the request and not as part of the URI. When the client uses the PUT method, it requests that the included entity body should be stored on the server at the requested URL.

DELETE Method

The DELETE method is used to delete the resource identified by the request URI. This method is used to delete resources remotely; however, authorization with processing of the DELETE method is required.

TRACE Method

The TRACE method allows programmers to see how a client's message is modified as it passes through a series of proxy servers. The recipient of the TRACE method echoes the HTTP request headers back to the client. When the TRACE method is used with the Max-Forwards and Via headers, a client can determine the chain of intermediate proxy servers between the original client and the server.

URL

An URL is a means of identifying a resource that is accessible through the Internet. The URL is a special case of a URI that is understood by web servers. A URL is any string that uniquely identifies an Internet resource.

Each URL is composed of three parts, a mechanism (or protocol) for retrieving the resource, the hostname of the server that can provide the resource, and a name for the resource. The resource name is usually a filename preceded by a partial path, which is relative to the path defined for the root of the web server. Here is an example:

http://www.cisco.com/en/US/support/index.html

In this example:

- The protocol is http; if no protocol is present, then most browsers default to http.
- The hostname or resource is www.cisco.com; this can be an IP address or a fully qualified domain name.
- The resource has a file called index.html; the path to the resource is en/US/support/.

HTTP Cookie

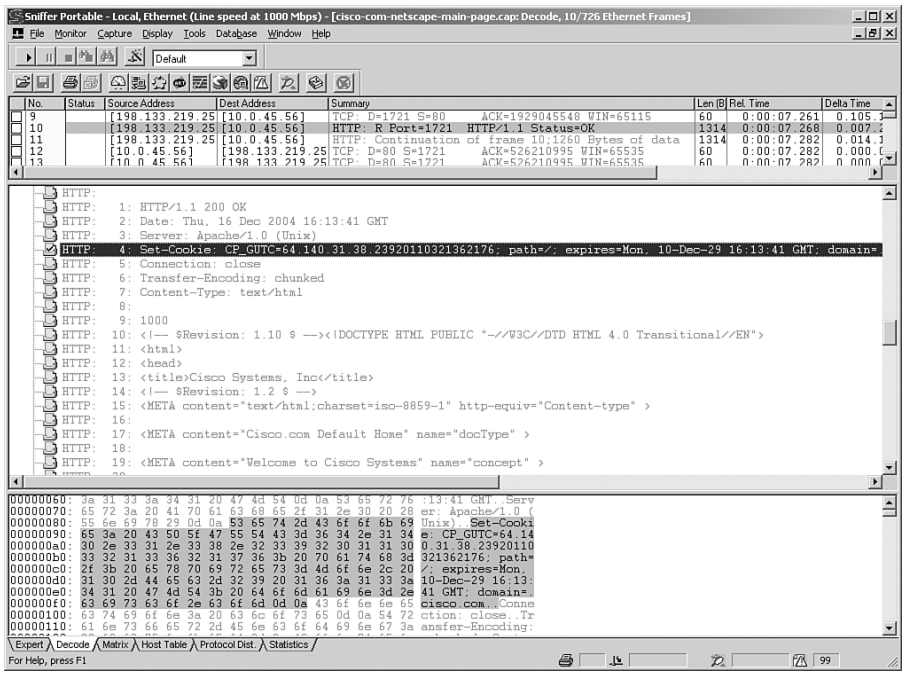
As the Internet has taken over the task of providing e-commerce applications to users, new requirements, such as maintaining a user to a specific server for a persistent connection, have become important. As mentioned earlier, HTTP is a stateless protocol where each HTTP request is independent of the other. For many applications, a server needs to track the user's request to send appropriate customized content to the user. This user tracking is extremely important for dynamic content, which provides user-specific information, such as the contents of a shopping cart. Earlier web servers tracked users by their IP addresses, but this became difficult as users starting connecting to servers from behind mega proxy links, such as AOL. In other words, an application cannot present the same shopping cart information for everyone using AOL.

Netscape resolved this issue by proposing the use of strings called cookies within HTTP headers. When the client sends the initial request to the server, the server returns a "Set-Cookie" header that gives a cookie name, expiry time, and other info. When the user

returns to the same URL, the user's browser returns the cookie if it has not expired. Cookies can be long lived or per-session based.

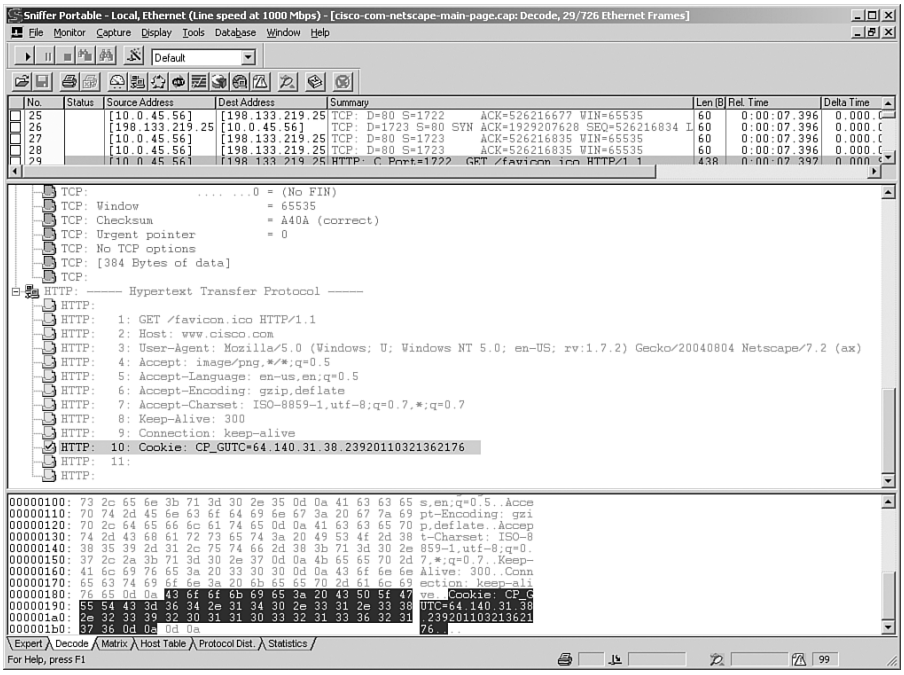
Cookies are simply text-based strings. When a client makes a connection to a web server, a cookie is inserted in the HTTP response from the server back to the client. An additional line with the Set-Cookie field is added. For example, Figure 4-6 shows a response from a server with an HTTP Set-Cookie header.

Figure 4-6 *Sniffer Capture of an HTTP Response with a Set-Cookie*



As the figure illustrates, the name of the cookie is the string "CP_GUTC," and the value for the cookie "CP_GUTC" is the numeric string that starts with "64.140." The path field indicates the particular directory within the site for which this cookie is valid. The path "/" indicates that the cookie is valid for all subdirectories and URL paths. The "expires" field specifies the validity time of the cookie. The client receives this cookie and stores it with the path and domain information for the period of time specified by the "expires" value. When the same client makes subsequent requests to the same URL, it uses the stored cookie in the HTTP GET. The Sniffer capture in Figure 4-7 shows how the client's second request has the cookie in the HTTP header.

Figure 4-7 *Sniffer Capture of the Second HTTP Request from the Client with the Cookie*



HTTP Cookie Parameters

As seen in the previous section, an HTTP cookie has six key parameters associated with it:

- The *name* and *value* of the cookie, a pair of strings used to identify the cookie and the value set for it. This is a mandatory header required by the Set-Cookie field.
- The *expiration date* of the cookie attribute specifies a date string that defines the validity of the cookie. The cookie is expired from the client browser and is no longer sent once the date has been reached. This is an optional header and is not required by the Set-Cookie field. If there is no expiration date sent from the server to the client, the cookie is considered a session cookie and is only valid for the length of the session to the URL. Once the client browser is closed, the cookie is not saved.
- The *path* the cookie sets is for the URL within which the cookie is valid. Pages outside of that path cannot read or use the cookie. If the path parameter is not set explicitly, then it defaults to the URL path of the document that the server is creating. If the path for the cookie is /test and the cookie value pair is test=ing123, the cookie will be sent along with the HTTP requests if those paths include /test/cgi-bin or /test/Cisco. As long as the /test is in the path, the cookie will be sent. The path "/" is the most general path and will send cookies to the sites associated with any HTTP requests the client issues.

- The *domain* value for the cookie is used when the server is doing a cookie match, and a comparison of the domain attributes of the cookie is made with the Internet domain name of the host from which the HTTP URL request is being requested. If a domain is matched, the request will be matched to the path of the request. The default value of the domain attribute is the domain of the server that generates the cookie response.
- The *secure* field specifies to transmit the cookie only if the connection is HTTPS, an SSL-encrypted connection. If secure is not specified, a cookie is considered safe to be sent in the clear text.

HTTP Header Fields

Headers are crucial to HTTP, as they determine the handling of a request. If a header is not recognized by the recipient, it should be ignored; if it is received by a proxy, it should be forwarded. HTTP (1.0) defines the following headers:

- General headers
- Request headers
- Response headers
- Entity headers

General Headers

General headers indicate general information, such as the date or whether the connection should be maintained. They are used by both clients and servers. The following are some of the general header fields:

- *The Date Header* in the corresponding response message indicates that the message was generated at the indicated time and has no bearing on when the associated entity may have been created or modified.
- *The Pragma* header permits directives to be sent to the recipient of the message. A directive is a way to request that components behave in a particular way while handling a request or a response.
- The *Connection* general header field allows the sender to specify options that are desired for that particular connection but that are not to be communicated by proxies over further connections.
- The *Trailer* general field value indicates that the given set of header fields is present in the trailer of a message encoded with chunked transfer coding.
- The *Transfer-Encoding* general header field indicates what (if any) type of transformation has been applied to the message body in order to safely transfer it between the sender and the recipient.

- The *Upgrade* general header allows the client to specify what additional communication protocols it supports and would like to use if the server finds it appropriate to switch protocols. The server must use the Upgrade header field within a 101 (Switching Protocols) response status code to indicate which protocol(s) are being switched.
- The *Upgrade* header field only applies to switching application-layer protocols upon the existing transport-layer connection. Upgrade cannot be used to insist on a protocol change; its acceptance and use by the server is optional. The capabilities and nature of the application-layer communication after the protocol change is entirely dependent upon the new protocol chosen, although the first action after changing the protocol must be a response to the initial HTTP request containing the Upgrade header field.
- The *Via* general header field must be used by gateways and proxies to indicate the intermediate protocols and recipients between the user agent and the server on requests, and between the origin server and the client on responses.
- The *Warning* general header field is used to carry additional information about the status or transformation of a message that might not be reflected in the message. This information is typically used to warn about a possible lack of semantic transparency from caching operations or transformations applied to the entity body of the message.

Request Headers

Request headers are used only for client's requests. They convey the client's configuration and desired format to the servers. Following are some of the request header fields:

- The *Authorization* header is used by the client to include the appropriate credentials required to access a resource. Certain resources cannot be accessed by the servers without proper authorization.
- The *From* header allows users to include their e-mail address as an identification. General use of the From header is discouraged, as it violates the privacy of the user.
- The *If-Modified-Since* header is a conditional header, indicating that the request may be handled in a different way based on the value specified in the header field.
- The *Referer* header lets the clients include the URI of the resource from which the request URI was obtained.
- The *Accept-Charset* request header field can be used to indicate what character sets are acceptable for the response. This field allows clients capable of understanding more comprehensive or special-purpose character sets to signal that capability to a server capable of representing documents in those character sets.
- The *Accept-Encoding* request header field is similar to Accept-Charset but restricts the content codings that are acceptable in the response. If an Accept-Encoding field is present in a request, and if the server cannot send a response that is acceptable according to the Accept-Encoding header, then the server should send an error response with the 406 (Not Acceptable) status code.

- The *Accept-Language* request header field is similar to Accept-Charset, but restricts the set of natural languages that are preferred as a response to the request.
- The *Authorization* field value consists of credentials containing the authentication information of the user agent for the realm of the resource being requested.
- The *Expect* request header field is used to indicate that particular server behaviors are required by the client.
- The *From* request header field, if given, should contain an Internet e-mail address for the human user who controls the requesting user agent.
- The *Host* request header field specifies the Internet host and port number of the resource being requested, as obtained from the original URI given by the client.
- The *If-Match* request header field is used with a method to make it conditional. A client that has one or more entities previously obtained from the resource can verify that one of those entities is current by including a list of their associated entity tags in the If-Match header field.
- The *If-Unmodified-Since* request header field is used with a method to make it conditional. If the requested resource has not been modified since the time specified in this field, the server should perform the requested operation as if the If-Unmodified-Since header were not present.
- The *Location* response header field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource.
- The *Max-Forwards* request header field provides a mechanism with the TRACE and OPTIONS methods to limit the number of proxies or gateways that can forward the request to the next inbound server. This can be useful when the client is attempting to trace a request chain that appears to be failing or looping in mid-chain.
- The *Proxy-Authorization* request header field allows the client to identify itself (or its user) to a proxy that requires authentication. The Proxy-Authorization field value consists of credentials containing the authentication information of the user agent for the proxy and/or realm of the resource being requested.
- The *Referer* request header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained (the "referrer," although the header field is misspelled.) The Referer request header allows a server to generate lists of back-links to resources for interest, logging, optimized caching, and so on. It also allows obsolete or mistyped links to be traced for maintenance.
- The *TE* request header field indicates what extension transfer-codings it is willing to accept in the response and whether it is willing to accept trailer fields in a chunked transfer coding.
- The *User-Agent* request header field contains information about the user agent originating the request. This is for statistical purposes, the tracing of protocol violations, and the automated recognition of user agents for the sake of tailoring responses to avoid particular user agent limitations.

Response Headers

Response headers are used only in server responses. They describe the server's configuration and information about the requested URL. The response headers start with the status line followed by the other request-initiated headers. The status line of the server's response includes the HTTP version number, a three-digit status code, and a textual description of the result.

HTTP defines a few specific codes in each range, although these ranges will become more populated as HTTP evolves. If a client cannot decipher a status code, it should be able to understand its basic meaning from its numerical range.

Following are some of the response header fields:

- *Status-Line* is the first line of a response message and consists of the protocol version followed by a numeric status code and its associated textual phrase. HTTP status codes are extensible. Following are key status code definitions from RFC 2616:
  - 1xx Informational
  - 100 Continue
  - 101 Switching Protocols
  - 2xx Successful
  - 200 OK
  - 201 Created
  - 202 Accepted
  - 203 Non-Authoritative Information
  - 204 No Content
  - 205 Reset Content
  - 206 Partial Content
  - 3xx Redirection
  - 300 Multiple Choices
  - 301 Resource Moved Permanently
  - 301 Resource Moved Temporarily
  - 303 See Other
  - 304 Not Modified
  - 305 Use Proxy
  - 306 (Unused)
  - 307 Temporary Redirect

  - 4xx Client Error
  - 400 Bad Request
  - 401 Unauthorized
  - 402 Payment Required
  - 403 Forbidden
  - 404 Not Found
  - 405 Method Not Allowed
  - 406 Not Acceptable
  - 407 Proxy Authentication Required
  - 408 Request Timeout
  - 409 Conflict
  - 410 Gone
  - 411 Length Required
  - 412 Precondition Failed
  - 413 Request Entity Too Large
  - 414 Request-URI Too Long
  - 415 Unsupported Media Type
  - 416 Requested Range Not Satisfiable
  - 417 Expectation Failed
  - 5xx Server Error
  - 500 Internal Server Error
  - 501 Not Implemented
  - 502 Bad Gateway
  - 503 Service Unavailable
  - 504 Gateway Timeout
  - 505 HTTP Version Not Supported
- The *Age* response header field conveys the sender's estimate of the amount of time since the response (or its revalidation) was generated at the origin server. A cached response is "fresh" if its age does not exceed its freshness lifetime.
- The *ETag* response header field provides the current value of the entity tag for the requested variant.
- The *Location* response header field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource.

- The *Proxy-Authenticate* response header field must be included as part of a 407 (Proxy Authentication Required) status code response. The field value consists of a challenge that indicates the authentication scheme and parameters applicable to the proxy for this Request-URI.
- The *Server* response header field contains information about the software used by the origin server to handle the request.
- The *Vary* field value indicates the set of request header fields that fully determines, while the response is fresh, whether a cache is permitted to use the response to reply to a subsequent request without revalidation.
- The *WWW-Authenticate* response header field must be included in 401 (Unauthorized) response messages. The field value consists of at least one challenge that indicates the authentication scheme(s) and parameters applicable to the Request-URI.

Entity Headers

Entity headers describe the document format of the data being sent between the client and the server. Although entity headers are most commonly used by the server when returning a requested document, they are also used by clients when using the POST and PUT methods. Following are some of the entity header fields:

- The *Content-Encoding* entity header field is used as a modifier to the media-type. When present, its value indicates what additional content codings have been applied to the entity body, and thus what decoding mechanisms must be applied in order to obtain the media type referenced by the Content-Type header field.
- The *Content-Language* entity header field describes the natural language(s) of the intended audience for the enclosed entity. Note that this might not be equivalent to all the languages used within the entity body.
- The *Content-Length* entity header field indicates the size of the entity body, in decimal number of OCTETs, sent to the recipient—or, in the case of the HEAD method, the size of the entity body that would have been sent had the request been a GET.
- The *Content-Location* entity header field may be used to supply the resource location for the entity enclosed in the message when that entity is accessible from a location separate from the requested resource's URI. A server should provide a Content-Location for the variant corresponding to the response entity. Especially in the case where a resource has multiple entities associated with it and those entities actually have separate locations by which they might be individually accessed, the server should provide a Content-Location for the particular variant returned.
- The *Content-MD5* entity header field, as defined in RFC 1864, is an MD5 digest of the entity body for the purpose of providing an end-to-end message integrity check (MIC) of the entity body. (Note that a MIC is good for detecting accidental modification of the entity body in transit, but it is not proof against malicious attacks.)

- The *Content-Range* entity header is sent with a partial entity body to specify where in the full entity body the partial body should be applied. The Content-Type entity header field indicates the media type of the entity body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.
- The *Expires* entity header field gives the date and time after which the response is considered stale. A stale cache entry cannot normally be returned by a cache (either a proxy cache or a user agent cache) unless it is first validated with the origin server (or with an intermediate cache that has a fresh copy of the entity).
- The *Last-Modified* entity header field indicates the date and time at which the origin server believes the variant was last modified.

Differences Between HTTP Versions 1.0 and 1.1

The following sections discuss some of the key differences between HTTP 1.0 and 1.1:

- Persistent connections
- Chunked messages
- Hostname
- Pipelining requests

Persistent Connections

Persistent connection is the main difference between version 1.0 and 1.1. HTTP 1.0, in its documented form, made no provision for persistent connections. Some HTTP 1.0 implementations use a Keep-Alive header to request that a connection persist.

HTTP 1.1 makes persistent connections by default. HTTP 1.1 clients, servers, and proxies assume that a connection will be kept open after the transmission of a request and its response. The protocol does allow an implementation to close a connection at any time, in order to manage its resources, although it is best to do so only after the end of a response.

Chunked Messages

HTTP 1.1 resolves the problem of delimiting message bodies by introducing the chunked transfer coding. The sender breaks the message body into chunks of arbitrary length, and each chunk is sent with its length prepended; it marks the end of the message with a zero-length chunk. The sender uses the transfer encoding chunked header to signal the use of chunking.

This mechanism allows the sender to buffer small pieces of the message, instead of the entire message, without adding much complexity or overhead. All HTTP 1.1 implementations must be able to receive chunked messages.

Hostname

HTTP 1.0 requests do not pass the hostname as part of the request URL. For example, if a user makes a request for the resource at URL http://www.cisco.com/index.html, the browser sends a message with the following request line to the server at www.cisco.com:

```
GET /index.html HTTP/1.0
```

This prevents the binding of another HTTP server hostname, such as exampleB.org to the same IP address, because the server receiving such a message cannot tell which server the message is meant for.

HTTP 1.1 requires that requests include a Host header that carries the hostname. This converts the preceding example to:

```
GET /index.html HTTP/1.1
Host: www.cisco.com
```

If the URL references a port other than the default (TCP port 80), this is also given in the Host header.

Pipelining Requests

Although HTTP 1.1 encourages the transmission of multiple requests over a single TCP connection, each request must still be sent in one contiguous message, and a server must send responses in the order that it received the corresponding requests. However, a client need not wait to receive the response for one request before sending another request on the same connection. In fact, a client could send an arbitrarily large number of requests over a TCP connection before receiving any of the responses. This practice, known as pipelining, can greatly improve performance. It avoids the need to wait for network round-trips, and it makes the best possible use of the TCP protocol.

Layer 7 Load Balancing Mechanisms

In the previous section, we covered the HTTP protocol in detail. You now understand the URLs, methods, and cookies. In Layer 7 load balancing, the SLB device proxies the client's TCP connection and receives the HTTP request. The SLB device buffers the client request and parses through it. The load balancer can perform many functions while inspecting the HTTP header. Following are some of the key mechanisms that can be used in Layer 7 load balancing:

- HTTP methods-based load balancing
- HTTP URL-based load balancing
- HTTP cookie-based load balancing
- HTTP cookie passive-based load balancing
- HTTP cookie learn-based load balancing

HTTP Methods-Based Load Balancing

The SLB device can definitely inspect the HTTP method used by the client and make appropriate load-balancing decisions. For instance, the SLB device can be configured to distribute GET and POST methods to separate server farms. The SLB device can easily drop DELETE method calls to prevent hackers from deleting web content.

HTTP URL-Based Load Balancing

The SLB device can inspect the HTTP URL and make appropriate load-balancing decisions. The device can distribute requests based on access content or application to different server farms. For example, all .cgi requests can be sent to servers optimized for request processing and computation, while all static content requests (.htm, .gif, and so on) can be sent to servers with a lot of disk space. Similarly, server management can be eased up by keeping separate server farms for /sports/* and /news/*. Figure 4-8 shows how a Layer 7 load-balancing device can be used to inspect HTTP requests and distribute client requests based on content type.

Figure 4-8 *Layer 7 SLB Used for Distributed Content*

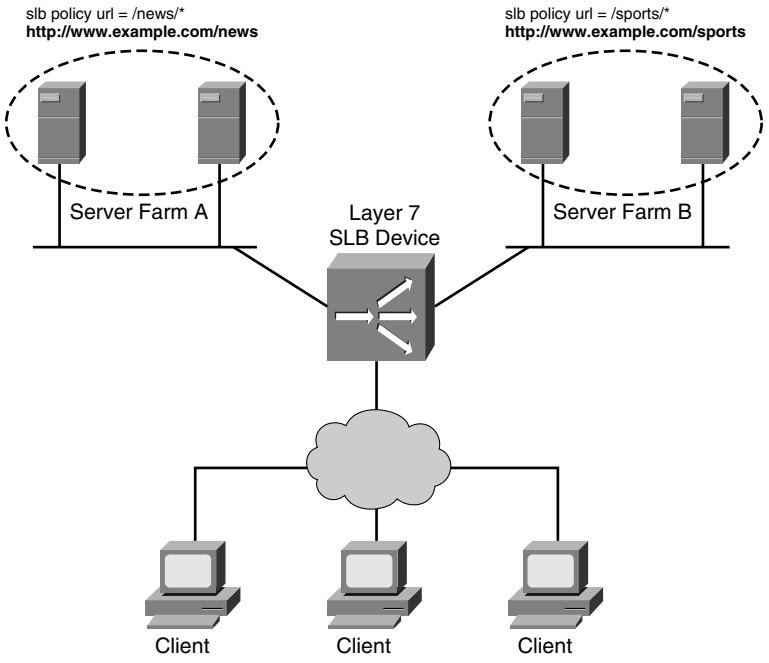
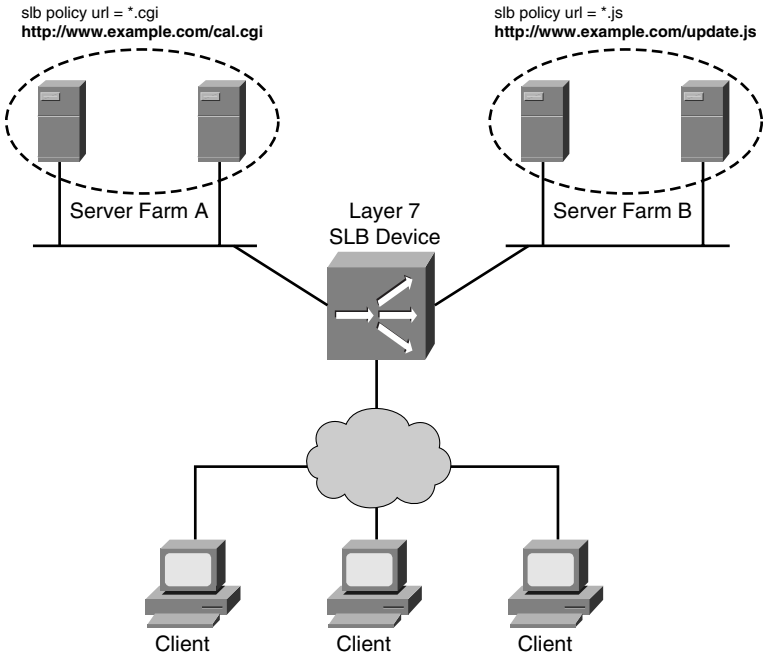


Figure 4-9 shows how a Layer 7 load-balancing device can be used to inspect HTTP requests and distribute client requests based on the applications being accessed.

Figure 4-9 *Layer 7 SLB Used for Distributed Applications*



HTTP Cookie-Based Load Balancing

The SLB device can make a load-balancing decision based on the clients' cookies. This is vital for applications, such as Web Logic, which uses multiple cookies to provide client session redundancy.

The Content Switching Module (CSM) has the "Cookie" expression matching feature that can filter for a specific cookie name and value. In this configuration, the CSM is not looking at the "cookie" inserted by the server; it is only matching the cookie in the client HTTP request. For example:

```
!
 map COOKIE_SERVER1 cookie
  match protocol http cookie CookieName cookie-value 101617
!
 map COOKIE_SERVER2 cookie
  match protocol http cookie CookieName cookie-value 101327
!
```

Each cookie map maps to a policy that has a single server farm. All the policies are added to the virtual. A default policy needs to be configured to make sure the user request is load balanced across the servers in case a cookie match is not found in the user HTTP GET. In cookie matching, the cookie name needs to be a specific name, while the value of the cookie can be a regular expression.

The functional steps of the cookie matching feature are as follows:

1. The user connects to the virtual server. The CSM proxies the TCP connect and waits for the HTTP GET.
2. The HTTP GET is received from the user, and then the CSM parses the GET and searches for the configured cookie names. If a match is found, the policy action is taken.
3. If a cookie match is not found, the default policy is used to load balance the user among the servers.

HTTP Cookie Passive-Based Persistence

When making persistent session decisions on cookies, the SLB device can read the cookie value for the specified named cookie that the server has installed and send the next request to the first server. This basically means that the load balancer reads the cookies set by the server. The HTTP cookie values from each server would need to be different and also to be configured on the SLB device.

HTTP Cookie Learn-Based Persistence

Cookie learn-based persistence (also known as cookie sticky) is similar to passive-based persistence except that in this mode the SLB device dynamically learns the cookie values set by the server for each client and stores them in the cookie sticky table.

The CSM supports the cookie sticky feature. It requires the real server to insert the cookie into the HTTP reply. Assume that you configure the real server to insert cookie *"USERID=<xyz>";* where "USERID" is the name of the cookie and "xyz" is a unique per-user value. The CSM can stick the user to a real server based on the value of cookie "USERID".

The cookie name is case sensitive. Configuration is fairly simple; for example:

```
module ContentSwitchingModule 9
 !
 serverfarm REALS
  nat server
  no nat client
  real 10.2.6.17
   inservice
  real 10.2.6.27
   inservice
 !
 sticky 10 cookie USERID timeout 60
 !
 vserver WWW
  virtual 10.2.44.56 tcp www
  serverfarm REALS
  sticky 60 group 10
  inservice
 !
```

Functional steps of the cookie sticky feature are as follows:

1. The user connects to the virtual server. The CSM proxies the TCP connect and waits for the HTTP GET.
2. The HTTP GET is received from the user, and then the CSM parses the GET and searches for the configured cookie name (USERID). If the cookie is found with a non-null value, that value is searched in the sticky table and the user is sent to the server that first issued the cookie.
3. If the cookie "USERID" is not found, the configured load-balancing method is used to select an available server—let's say real 10.2.6.17 is selected.
4. Real server 10.2.6.17 sends an HTTP response. The CSM parses through the response and searches for the "USERID" cookie. If there is a set cookie (let's say USERID=znaseh), then the CSM adds that to the sticky table. Thus the sticky table looks like this:

```
znaseh ---- 10.2.6.17
```

5. Next time the client comes in with USERID=znaseh, the request would be sent to server 10.2.6.17.

HTTP Cookie Insert-Based Persistence

The current load balancer can even insert cookies in the HTTP headers to provide persistence. This is a unique feature where the SLB device adds a cookie (name and value) into the HTTP response from the server. The client accepts the cookie as being sent by the server when in fact it comes from the SLB device.

The CSM supports the cookie insert feature. This feature is used when you want to use a session cookie for persistence if the server is not currently setting the appropriate cookie. With this feature enabled, the CSM inserts the cookie in the response from the server to the client.

The following example shows how to specify a cookie insert feature for session persistence:

```
module ContentSwitchingModule 9
!
sticky 25 cookie USERID insert timeout 30
!
 vserver WWW
  virtual 10.2.44.56 tcp www
  serverfarm REALS
  sticky 30 group 25
  inservice
 !
```

Case Study: Layer 7–Based Solution

Now that we understand TCP and HTTP protocol definitions and Layer 7 load-balancing functionality, we will start our discussion of how this solution should be deployed to solve various application scalability and security needs. Deploying a Layer 7–based

load-balancing solution requires extensive planning and research on how the application functions.

In this case study, we will examine a specific customer deployment where three different applications are being deployed: an online download application, an online shop application, and an online user profile application. We will show how each application's requirements are achieved by our Layer 7 load-balancing solution. The product of choice used in this solution is the CSM on the Cisco Catalyst 6500 Series Switch. The idea behind the following case study and others in chapters in the rest of the book is to understand not only the concepts but the implementation details.

In this case study, we will focus on server and application requirements. The management and security requirements as well as the infrastructure requirements stay the same as those defined in the Chapter 3 case study.

Server and Application Requirements

Server and application requirements for this case study are defined as follows:

- Directly manage the servers from the front end (client-side network)
- Number of servers 20—100 per site
- Following three different applications with their own distinct requirements
- Online Download Services Application
  - This custom application provides web-based software download capabilities to the clients.
  - Use TCP server port 80 for the client connections initiated to the server.
  - Clients can be distinguished by the URLs they use. Partner and premium clients should be sent to the high-end server farm. This ensures that the premium customers and partners get expedited services.
  - Source IP–based stickiness is needed for this application for up to 20 minutes.
- Online Shop Application
  - This custom application provides web-based software purchase capabilities to the client.
  - TCP/80, TCP/443
  - This is a secure application and only allowed on TCP port 443. A client that sends requests on HTTP (TCP:80) should be redirected to come back on the HTTPS site.
  - Source IP–based stickiness is needed for this application for up to 30 minutes.

- Online User Profile Application
  - This custom application provides web-based user profile update capabilities to the client.
  - TCP/80.
  - This application provides different features to internal and external users. The internal and external users should be sent to different servers. The internal and external users will use different domain names to reach the application.
  - No stickiness is required for this application.
- TCP-based keep alive required for all the application servers
- Potentially, server persistence needed when user transitions from HTTP to HTTPS

Infrastructure Configuration

In the Layer 7 SLB design, the CSM is used in routed mode with one client-side VLAN (12) and one server-side VLAN (112). The CSM's default gateway is the HSRP group IP on the MSFC, and the server's default gateway is the alias IP address on the CSM.

Following is the infrastructure configuration of the CSM, showing the client- and server-side VLANs. Notice that there are two VLANs defined. The client VLAN 12 is the client-facing VLAN, which also has a gateway defined (acting as the default gateway for the CSM) and the server VLAN 112, which is the server-facing VLAN. These VLANs are part of the port-channel from the CSM to the Supervisor (the Layer 2 Switch on the Cisco Catalyst 6500 Series Switch). Traffic will be forwarded from the Client VLAN 12 and will hit a vserver; from there it will be forwarded to the server VLAN 112 to be forwarded to the server. The alias address is the virtual address that is shared between the active and the standby CSM. The alias IP address shares a virtual MAC (Layer 2) address between the active and standby, and only the active CSM responds to requests for the virtual MAC address.

```
module ContentSwitchingModule 9
 vlan 12 client
  ip address 10.2.10.5 255.255.255.0
  gateway 10.2.10.1
  alias 10.2.10.4 255.255.255.0
!
 vlan 112 server
  ip address 10.2.12.2 255.255.255.0
  alias 10.2.12.1 255.255.255.0
!
```

Probe Configuration

Following are the probes that would be used to provide the health check of the applications. Notice the two kinds of probes that are defined (HTTP and TCP). The HTTP probe issues an HTTP GET request to the server under the server farm HTTP (under the server farm

configuration for the real servers 10.2.12.27 and 10.2.12.28), and it expects to get a 200 OK status back for it to keep the real server operational and for it to be forwarded connections from the client. Similarly, for the server farm TCP, the probe TCP will issue a TCP SYN to the real servers, and expects to receive a SYN-ACK back for the real servers to be operational and for it to be forwarded connections from the client. The frequencies for the probes to check the status of the real servers are defined by keyword interval (under the probe configuration) in seconds. The interval is the time in between probes when the real server is operational. The keyword **retries** is an integer used to set the number of failed probes that are allowed before marking the server as nonoperational. The **failed** keyword is used to set the time in seconds, in between probes when the real servers have been marked as nonoperational. The port numbers for these probes, if not defined under the probe configurations, are inherited from the server farm; if not defined under the server farm, the port numbers are inherited from the vserver. For the probe to be activated, the vserver (which has the server farm configuration) needs to be activated.

```
!
 probe HTTP http
  request method get url /keepalive.html
  expect status 200
  interval 5
  failed 5
!
 probe TCP tcp
  interval 5
  retries 2
  failed 5
!
serverfarm HTTP
  nat server
  no nat client
  real 10.2.12.27
   inservice
  real 10.2.12.28
   inservice
  probe HTTP
!
serverfarm TCP
  nat server
  no nat client
  real 10.2.12.27
   inservice
  real 10.2.12.28
   inservice
  probe TCP
!
```

Online Download Application

Following is the solution that meets the custom Online Download Application requirements. Notice how the URL maps capture partner and premium client requests. Clients without the partner and premium in their URLs are sent to the default server farm.

```
!
 map ONL_URLS url
  match protocol http url /partner/*
  match protocol http url /premium/*
```

```
!
sticky 11 netmask 255.255.255.255 timeout 20
sticky 12 netmask 255.255.255.255 timeout 20
!
 serverfarm ONL_80_DEF
  nat server
  no nat client
  real 10.2.12.27
   inservice
  real 10.2.12.28
   inservice
  probe TCP
!
 serverfarm ONL_80_URL
  nat server
  no nat client
  real 10.2.12.17
   inservice
  real 10.2.12.18
   inservice
  probe TCP
!
!
 policy ONL_80_URL
  url-map ONL_URLS
  sticky-group 11
  serverfarm ONL_80_URL
!
 policy ONL_80_DEF
  sticky-group 12
  serverfarm ONL_80_DEF
!
 vserver V_ONL_80
  virtual 10.2.10.101 tcp www
  vlan 12
  replicate csrp connection
  persistent rebalance
  slb-policy ONL_80_URL
  slb-policy ONL_80_DEF
  inservice
!
!
```

Online Shop Application

The Online Shop Application solution is shown here. Notice how the *webhost relocation* feature is used to send an HTTP 302 to the client when it arrives on clear text TCP port 80.

```
!
 serverfarm SHOP_443
  nat server
  no nat client
  real 10.2.12.57
   inservice
  real 10.2.12.58
   inservice
!
 serverfarm SHOP_80
  nat server
  no nat client
```

```
  redirect-vserver REDIRECT
   webhost relocation https://shop.example.com/
   inservice
!
!
sticky 13 netmask 255.255.255.255 timeout 30
!
 policy SHOP_80
  serverfarm SHOP_80
!
 policy SHOP_443
  sticky-group 13
  serverfarm SHOP_443
!
 vserver SHOP_443
  virtual 10.2.10.105 tcp https
  vlan 12
  replicate csrp sticky
  replicate csrp connection
  persistent rebalance
  slb-policy SHOP_443
  inservice
!
 vserver REDIRECT
  virtual 10.2.10.105 tcp www
  vlan 12
  persistent rebalance
  slb-policy SHOP_80
  inservice
!
```

Online User Profile Application

The Online User Profile Application requirements are met by the following solution. Notice how the http field and header information is specified by the USER_EXT map for the profile.example.com domain and the USER_INT header for the inprofile.example.com domain. Under the vserver, both the policies are applied and will be checked in linear order (that is, the USER_INT policy will be checked first and then the USER_EXT policy). Once a match is found based on the HTTP HOST header value, the real servers for the server farms defined under the policy will be forwarded the client request.

```
!
!
 map USER_EXT header
  match protocol http header Host header-value profile.example.com
!
 map USER_INT header
  match protocol http header Host header-value inprofile.example.com
!
 serverfarm USER_EXT
  nat server
  no nat client
  real 10.2.12.47
   inservice
  real 10.2.12.48
   inservice
  probe TCP
!
 serverfarm USER_INT
  nat server
```

```
  no nat client
  real 10.2.12.57
   inservice
  real 10.2.12.58
   inservice
  probe TCP
!
!
 policy USER_EXT
  header-map USER_EXT
  serverfarm USER_EXT
!
 policy USER_INT
  header-map USER_INT
  serverfarm USER_INT
!
 vserver V_ONL_80
  virtual 10.2.10.107 tcp www
  vlan 12
  replicate csrp connection
  persistent rebalance
  slb-policy USER_INT
  slb-policy USER_EXT
  inservice
!
```

Maximum HTTP Request Parse Length

The key to note in the following captures is the maximum parse length. The parse length is the number of bytes the CSM will look through in an HTTP request in search of the configured URLs or cookies. If the configured data is not found within the parse length, the client request is discarded.

```
CAT-Native-4#show module contentSwitching 9 vservers name V_ONL_80 detail
V_ONL_80, type = SLB, state = OUTOFSERVICE, v_index = 14
  virtual = 10.2.10.101/32:80 bidir, TCP, service = NONE, advertise = FALSE
  idle = 3600, replicate csrp = connection, vlan = 12, pending = 30, layer 7
  max parse len = 2000, persist rebalance = TRUE
  ssl sticky offset = 0, length = 32
  conns = 0, total conns = 0
  Policy          Tot matches  Client pkts  Server pkts
  ------------------------------------------------------
  ONL_80_URL      0            0            0
  ONL_80_DEF      0            0            0
  (default)       0            0            0
CAT-Native-4#
```

Notice next how the parse length can be increased up to 4000 bytes from within the virtual server configuration mode.

```
CAT-Native-4#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
CAT-Native-4(config)#
CAT-Native-4(config)#module contentSwitchingModule 9
CAT-Native(config-module-csm)# vserver V_ONL_80
CAT-Nativ(config-slb-vserver)#
CAT-Nativ(config-slb-vserver)#parse-length ?
  <1-4000>  maximum number of bytes to parse
CAT-Nativ(config-slb-vserver)#parse-length 4000
CAT-Nativ(config-slb-vserver)#exit
```

```
CAT-Native(config-module-csm)#exit
CAT-Native-4(config)#exit
CAT-Native-4#
CAT-Native-4#show module contentSwi 9 vservers name V_ONL_80 detail
V_ONL_80, type = SLB, state = OUTOFSERVICE, v_index = 14
  virtual = 10.2.10.101/32:80 bidir, TCP, service = NONE, advertise = FALSE
  idle = 3600, replicate csrp = connection, vlan = 12, pending = 30, layer 7
  max parse len = 4000, persist rebalance = TRUE
  ssl sticky offset = 0, length = 32
  conns = 0, total conns = 0
  Policy          Tot matches  Client pkts  Server pkts
  ----------------------------------------------------
  ONL_80_URL      0            0            0
  ONL_80_DEF      0            0            0
  (default)       0            0            0
CAT-Native-4#
```

Parse length can also be increased globally by adjusting the MAX_PARSE_LEN_MULTIPLIER global variable.

```
CAT-Native-4#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
CAT-Native-4(config)#
CAT-Native-4(config)#
CAT-Native-4(config)#module contentSwitchingModule 9
CAT-Native(config-module-csm)#
CAT-Native(config-module-csm)#vserver V_ONL_80
CAT-Nativ(config-slb-vserver)#parse-length 1500
CAT-Nativ(config-slb-vserver)#exit
CAT-Native(config-module-csm)#variable MAX_PARSE_LEN_MULTIPLIER 3
CAT-Native(config-module-csm)#exit
CAT-Native-4(config)#exit
CAT-Native-4#
CAT-Native-4#show module contentSwitching 9 vservers name V_ONL_80 detail
V_ONL_80, type = SLB, state = OUTOFSERVICE, v_index = 14
  virtual = 10.2.10.101/32:80 bidir, TCP, service = NONE, advertise = FALSE
  idle = 3600, replicate csrp = connection, vlan = 12, pending = 30, layer 7
  max parse len = 4500, persist rebalance = TRUE
  ssl sticky offset = 0, length = 32
  conns = 0, total conns = 0
  Policy          Tot matches  Client pkts  Server pkts
  ----------------------------------------------------
  ONL_80_URL      0            0            0
  ONL_80_DEF      0            0            0
  (default)       0            0            0
CAT-Native-4#
CAT-Native-4#
```

CSM Configuration

The following is the completed CSM configuration:

```
module ContentSwitchingModule 9
 vlan 12 client
  ip address 10.2.10.5 255.255.255.0
  gateway 10.2.10.1
  alias 10.2.10.4 255.255.255.0
!
 vlan 112 server
  ip address 10.2.12.2 255.255.255.0
  alias 10.2.12.1 255.255.255.0
```

```
!
variable MAX_PARSE_LEN_MULTIPLIER 3
!
 map ONL_URLS url
  match protocol http url /partner/*
  match protocol http url /premium/*
!
 map USER_EXT header
  match protocol http header Host header-value profile.example.com
!
 map USER_INT header
  match protocol http header Host header-value inprofile.example.com
!
 probe HTTP http
  request method get url /keepalive.html
  expect status 200
  interval 5
  failed 5
!
 probe TCP tcp
  interval 5
  retries 2
  failed 5
!
sticky 11 netmask 255.255.255.255 timeout 20
sticky 12 netmask 255.255.255.255 timeout 20
sticky 13 netmask 255.255.255.255 timeout 30
!
 serverfarm ONL_80_DEF
  nat server
  no nat client
  real 10.2.12.27
   inservice
  real 10.2.12.28
   inservice
  probe TCP
!
 serverfarm ONL_80_URL
  nat server
  no nat client
  real 10.2.12.17
   inservice
  real 10.2.12.18
   inservice
  probe TCP
!
 serverfarm SHOP_443
  nat server
  no nat client
  real 10.2.12.57
   inservice
  real 10.2.12.58
   inservice
!
 serverfarm SHOP_80
  nat server
  no nat client
  redirect-vserver REDIRECT
   webhost relocation https://shop.example.com/
   inservice
!
 serverfarm USER_EXT
  nat server
  no nat client
```

```
  real 10.2.12.47
   inservice
  real 10.2.12.48
   inservice
  probe TCP
!
 serverfarm USER_INT
  nat server
  no nat client
  real 10.2.12.57
   inservice
  real 10.2.12.58
   inservice
  probe TCP
!
 policy ONL_80_URL
  url-map ONL_URLS
  sticky-group 11
  serverfarm ONL_80_URL
!
 policy ONL_80_DEF
  sticky-group 12
  serverfarm ONL_80_DEF
!
policy SHOP_80
  serverfarm SHOP_80
!
 policy SHOP_443
  sticky-group 13
  serverfarm SHOP_443
!
policy USER_EXT
  header-map USER_EXT
  serverfarm USER_EXT
!
 policy USER_INT
  header-map USER_INT
  serverfarm USER_INT
!
 vserver V_ONL_80
  virtual 10.2.10.101 tcp www
  vlan 12
  replicate csrp connection
  persistent rebalance
  slb-policy ONL_80_URL
  slb-policy ONL_80_DEF
  inservice
!
vserver SHOP_443
  virtual 10.2.10.105 tcp https
  vlan 12
  replicate csrp sticky
  replicate csrp connection
  persistent rebalance
  slb-policy SHOP_443
  inservice
!
 vserver REDIRECT
  virtual 10.2.10.105 tcp www
  vlan 12
  persistent rebalance
  slb-policy SHOP_80
  inservice
!
```

```
vserver V_ONL_80
  virtual 10.2.10.107 tcp www
  vlan 12
  replicate csrp connection
  persistent rebalance
  slb-policy USER_INT
  slb-policy USER_EXT
  inservice
!
```

Test and Verification

Test and verification of the Layer 7 application load-balanced environment is a lot more critical than for Layer 4 SLB solutions. This is because in the case of the Layer 7 load-balancing solution the SLB device has to proxy the client TCP connect and to buffer and search through the HTTP request. Each application is unique with respective to the number of sockets used, duration of TCP connections, activity in each session in terms of packets per second, idle timeouts, the size and number of the HTTP cookie, the TCP MSS value used, and so on. Thus, it is critical to test and verify the Layer 7 SLB environment with the particular applications.

Following are a few critical test cases that should be verified after a new deployment or a major or minor infrastructure change.

- Verifying and testing an exact trace of a production client load-balanced session.
- Verifying and testing a completed transaction with authentication and authorization from the client to the server.
- Verifying and testing a server-initiated session to a back-end database or internal application server.
- Verifying and testing a server-initiated backup or data replication session.
- Verifying and testing an application daemon failure and detection by the load balancer.
- Testing with clients using different browser types, such as Netscape, MS IE, Opera, Mozilla Firefox, and so on.
- Testing with clients coming in from behind different service providers and different service types, such as DSL, Cable broadband, wireless, dial-up, AOL, and so on.

Summary

This chapter introduced the Layer 7 load-balancing technology. The chapter covered the TCP and HTTP protocols and defined the protocol header fields and how they can be used in scaling applications. The TCP connection establishment and connection termination via the TCP SYN and SYN-ACK acknowledgements was discussed in detail. Similarly, based on the TCP handshakes between the client and the server, the HTTP Request and Response

headers were analyzed. The HTTP Cookie solution and how the URLs are handled by the HTTP protocol brought to light how load balancers handle the implementation of these HTTP features. The enhancements of persistent connections and having hostnames in HTTP 1.1 were reviewed. Several Layer 7 solutions, such as URL or HTTP header-based load balancing were also presented. In order to combine all the concepts of Layer 7 SLB from the perspective of the CSM platform, this chapter also presented a case study. This case study focused on the deployment scenario of a real-world enterprise solution where three distinct applications were being deployed. The applications illustrated the CSM Layer 7 load-balancing functionality via the use of HTTP URLs and HEADERS to match specific client connections to specific servers. The infrastructure setup for the CSM client and server VLANs and how the CSM operates probes to the server farms were reviewed. CSM configurations were provided to introduce the reader to the Layer 7 configuration CLI.

With the increased demand for security and for security devices to perform at greater speeds, Chapter 5 discusses firewall load balancing.