CHAPTER 1

Application Acceleration and Optimization Overview

This chapter describes how you can use application acceleration and optimization on the Cisco 4700 Series Application Control Engine (ACE) appliance to accelerate enterprise applications that results in increased employee productivity, enhanced customer retention, and increased online revenues.

This chapter describes the different application acceleration and optimization functions of the ACE. It contains the following major sections:

- AppScope Performance Monitoring Using the Cisco AVS 3180A Management Station
- Detailed ACE-Client Interaction
- FlashForward Operation
- Where to Go Next

Note Application acceleration performance on the ACE is 50 to 100 Mbps throughput. With typical page sizes and browser usage patterns, this equates to roughly 1,000 concurrent connections. Subsequent connections bypass the application acceleration engine. This limitation applies only to traffic that is explicitly configured to receive application acceleration processing (for example, FlashForward, Delta Optimization). Traffic that is not configured to receive application acceleration processing is not subject to these limitations. Also, because the ACE HTTP compression is implemented separately in hardware, it is not subject to these limitations. For example, if you have a mix of application-accelerated and non-application-accelerated traffic, the former is limited; the latter is not. If you have 50 Mbps of application-accelerated traffic, the ACE can still deliver up to 1.9 Gbps throughput for the non-application-accelerated traffic.

Optimization Traffic Policies and Typical Configuration Flow

To define the different application acceleration and optimization functions that you want the ACE to perform, as described in the sections in this chapter, you must configure the following software components:

- Optimization HTTP action list
- Optimization HTTP parameter map
- Layer 7 HTTP optimization policy map
- Layer 7 server load-balancing class map and policy map
- Layer 3 and Layer 4 server load-balancing class map and policy map
- Layer 3 and Layer 4 optimization policy map

The steps outlined below serve as a general overview on the configuration processes that you follow to implement the different application acceleration and optimization functions for the ACE:

Step 1 Create an optimization HTTP action list by using the **action-list type optimization http** command. The optimization HTTP action list groups together a series of individual application acceleration and optimization operations that you want the ACE to perform. For example, you can specify the following optimization functions in one or more actions lists:

- Delta optimization
- FlashForward
- FlashForward object acceleration
- Just-in-time object acceleration
- Adaptive dynamic caching
- Cache optimization

See Chapter 2, Configuring an Optimization HTTP Action List for details.

Step 2 (Optional) Develop an optimization HTTP parameter map by using the **parameter-map type optimization http** command. The optimization HTTP parameter map identifies the application acceleration and optimization parameters that adjust or control several optimization technologies based on the selections made in an associated action list. For example, you can configure the following optimization functions in one or more parameter maps depending on the functions specified in the associated action list:

- FlashForward parameters
- Basefile parameters
- Cache parameters
- Cache-policy parameters
- Canonical-url string
- Delta optimization modes and parameters
- Script language
- Freshness period of objects in the client browser
- Rebase parameters

- Load threshold for change in cache-ttl

See Chapter 3, Configuring an Optimization HTTP Parameter Map for details.

Step 3 Develop a Layer 7 server load balancing (SLB) class map and associate it with a Layer 7 policy map. The Layer 7 SLB class map and policy map act as a filter for traffic that matches the SLB criteria that you specify, such as a cookie, HTTP header, URL, or source IP address.

Note You can instruct the ACE to compress and encode packets that match a Layer 7 SLB policy map by using the **compress** command in policy map load-balancing class configuration mode. You define the compression format that the ACE uses when responding to an HTTP compression request from a client. For details, see the *Cisco 4700 Series Application Control Engine Appliance Server Load-Balancing Configuration Guide*.

See "Configuring a Layer 7 Class Map and Policy Map for SLB" in Chapter 4, Configuring a Traffic Policy for HTTP Optimization for details.

Step 4 Develop a Layer 7 optimization HTTP policy map by using the **policy-map type optimization http first-match**. The optimization HTTP policy map activates an optimization HTTP action list that you can use to configure the specified application acceleration and optimization actions. You can specify an optional optimization HTTP parameter list in an optimization HTTP policy map to identify the association between the action list and the parameter map. In this case, the optimization HTTP action list defines what to do while the optimization HTTP parameter map defines the specific details about how to accomplish the application acceleration action.

See "Configuring a Layer 7 Class Map and Policy Map for SLB" in Chapter 4, Configuring a Traffic Policy for HTTP Optimization for details.

Step 5 Develop a Layer 3 and Layer 4 class map. The Layer 3 and Layer 4 class map contains match criteria to classify network traffic that can pass through the ACE, such as virtual IP (VIP) address, the protocol, and the ACE port. The ACE uses these Layer 3 and Layer 4 traffic classes to perform SLB.

See "Configuring a Layer 3 and Layer 4 Class Map for SLB" in Chapter 4, Configuring a Traffic Policy for HTTP Optimization for details.

Step 6 Create a Layer 3 and Layer 4 policy map that contains SLB actions that are related to a VIP. In addition, you configure the application acceleration and optimization services to be performed by the ACE. This process binds the specified functions in the associated HTTP optimize action lists and parameter maps with the specified VIP.

See "Configuring a Layer 3 and Layer 4 Policy Map for SLB and Application Acceleration" in Chapter 4, Configuring a Traffic Policy for HTTP Optimization for details.

Step 7 Activate the Layer 3 and Layer 4 policy map by associating it with a specific VLAN interface or globally with all VLAN interfaces by using the **service-policy** command as the means to filter traffic received by the ACE.

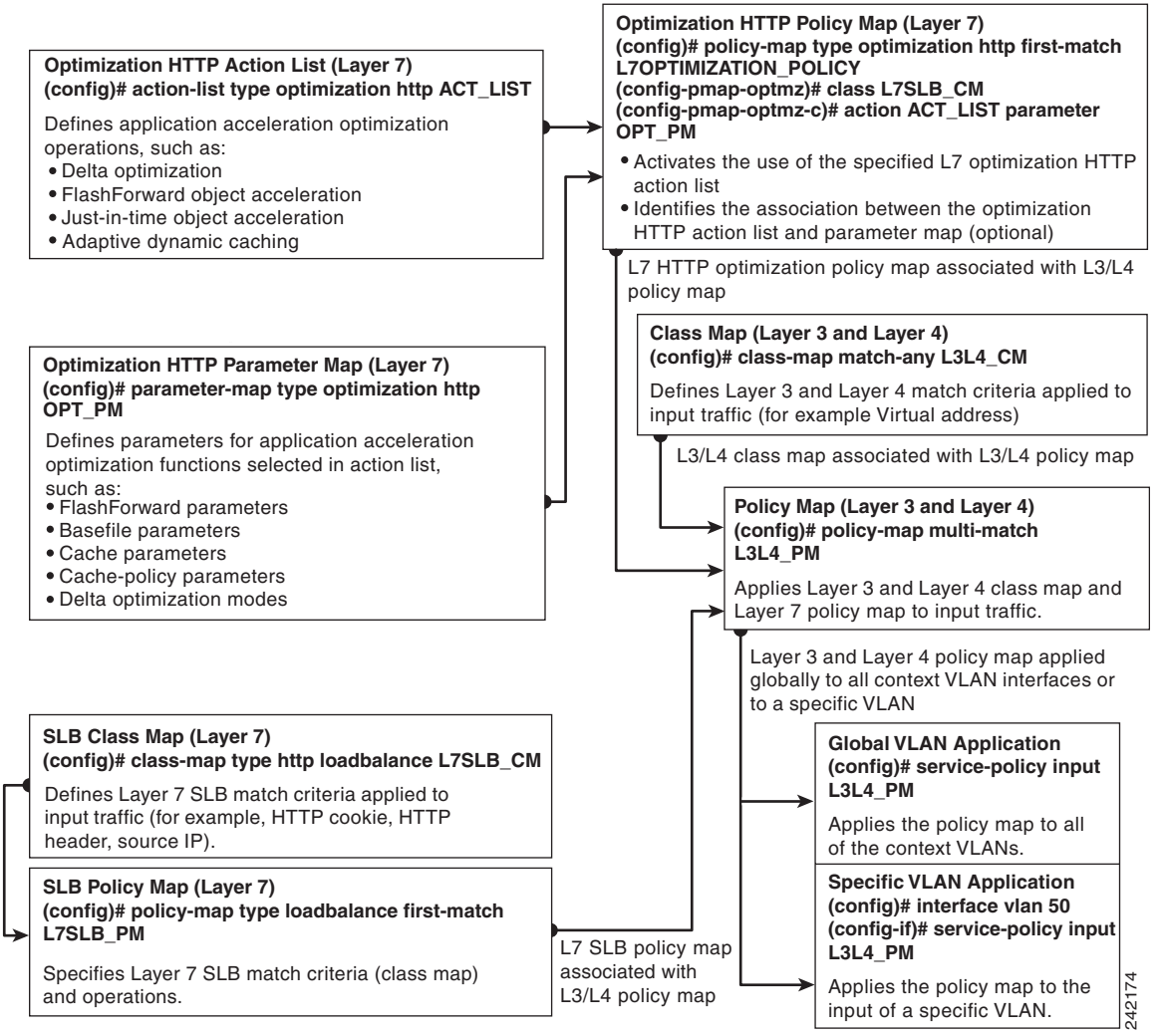
See "Applying a Service Policy" in Chapter 4, Configuring a Traffic Policy for HTTP Optimization for details.

Step 8 Define optimization parameters by using the **optimize** command. You apply specific optimization parameters on a global level to perform activities such as uploading application acceleration and optimization statistical log information to the optional Cisco AVS 3180A Management Station or inserting a global prefix in embedded object URLs before the host name.

See Chapter 5, Configuring Global Optimization Settings for details.

Figure 1-1 provides a basic overview of the process required to build and apply the Layer 7, Layer 3, and Layer 4 actions lists, parameter maps, and traffic policies that the ACE uses for SLB and application acceleration. The figure also shows how you associate the various components of the policy configuration with each other.

Figure 1-1 Application Acceleration and Optimization Configuration Flow Diagram



The following configuration example shows a running configuration that implements the different application acceleration and optimization functions for the ACE.

```
# ===================================
```

```
# Backend web server farm
# ====================================
rserver SERVER1
   ip address 192.168.10.100
   inservice

rserver SERVER2
   ip address 192.168.10.101
   inservice

serverfarm SFARM1
   rserver SERVER1 80
      inservice
   rserver SERVER2 80
      inservice

policy-map type loadbalance first-match L7_SLB_POLICY
   match any_url  http url .*
   serverfarm SFARM1

# ================================================
# Application Acceleration-Oriented Configuration
# ================================================
class-map type http loadbalance match-any L7_BYPASS_CLASS
   match http url .*\acc_post\.html
   match http url .*\acc_nopost\.html
   match http url .*\acc_dbgtrace.*.js
   match http url .*\acc_appscope.*.js

class-map type http loadbalance match-any L7_FLASHFORWARD_CLASS
   match http url .*\.gif
   match http url .*\.css
   match http url .*\.js
   match http url .*\.class
   match http url .*\.jar
   match http url .*\.cab
   match http url .*\.txt
   match http url .*\.ps
   match http url .*\.vbs
   match http url .*\.xsl
   match http url .*\.xml
   match http url .*\.pdf
   match http url .*\.swf

class-map type http loadbalance match-any L7FLASHFORWARD-IMG-OPT_CLASS
   match http url .*\.jpg
   match http url .*\.jpeg
   match http url .*\.jpe
```

```
   match http url .*\.png

class-map type http loadbalance match-any L7_CM-SLB_CLASS
   match http url .*\.html
   match http url .*\.htm

class-map type http loadbalance L7_DELTA_CLASS
   match http url .*

parameter-map type optimization http L7_FLASHFORWARD_POLICY
   cache ttl min 0
   cache ttl max 60

action-list type optimization http FLASHFORWARD-OBJ_AL
   flashforward-object

action-list type optimization http FLASHFORWARD-IMG-OPT_AL
   flashforward-object

action-list type optimization http DELTA_AL
   delta
   flashforward

action-list type optimization http FLASHFORWARD_AL
   flashforward


policy-map type optimization http first-match L7_OPTM_POLICY
   class L7_FLASHFORWARD_CLASS
      action FLASHFORWARD-OBJ_AL parameter L7_FLASHFORWARD_POLICY
   class L7FLASHFORWARD-IMG-OPT_CLASS
      action FLASHFORWARD-IMG-OPT_AL parameter L7_FLASHFORWARD_POLICY
   class L7_CM-SLB_CLASS
      action FLASHFORWARD_AL
   class L7_DELTA_CLASS
      action DELTA_AL


# ================================
# LAYER 4 CONFIGURATION
# ================================
class-map match-any L4_VIP_CLASS
   match virtual-address 172.16.2.142 any

policy-map multi-match L4_OPTIMIZE-VIP
   class L4_VIP_CLASS
      loadbalance policy L7_SLB_POLICY
      loadbalance vip inservice
```

```
      optimize http policy L7_OPTM_POLICY

#
# Bind with VLAN
#
interface gigabitEthernet 1/2
   channel-group 2
   no shutdown

interface gigabitEthernet 1/3
   channel-group 3
   no shutdown

interface port-channel 2
   switchport access vlan 2
   no shutdown

interface port-channel 3
   switchport access vlan 3
   no shutdown

access-list ACL1 line 10 extended permit ip any any

interface vlan 2
   ip address 172.16.2.141 255.255.255.0
   access-group input ACL1
   service-policy input OPTIMIZE-VIP
   no shutdown

interface vlan 3
   ip address 192.168.10.141 255.255.255.0
   access-group input ACL1
   no shutdown
```

Delta Optimization

The ACE enables enterprises to dynamically update client browser caches directly with content differences, or deltas. The results are faster page downloads, improved employee productivity, and increased online revenues.

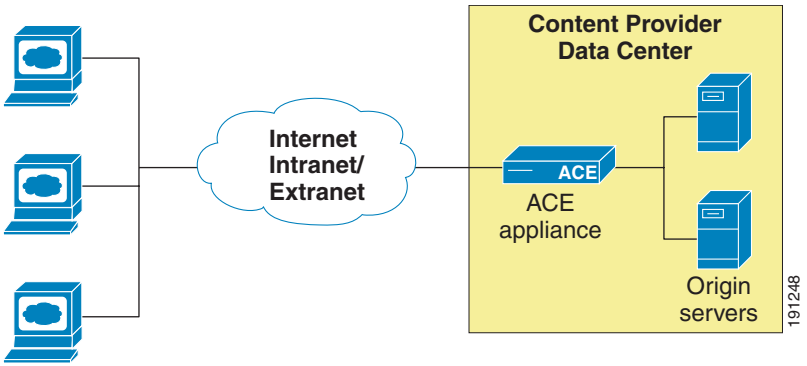
Many web pages are created dynamically so that each request produces different content. The differences in content could result from different payloads that are rotated in and out of the page or from more deeply rooted HTML content changes (for example, changing stock quotes, changing news headlines, and so on). Because these dynamically created pages change with each request, they are not cacheable by traditional caching solutions. Even with web caching, users must download the entire HTML document from the origin server each time that they request the document, even though the differences in each subsequent download are small compared to the size of the entire page.

For example, a news service home page consists of 70 KB of HTML. This page is dynamic because it contains different payloads for each request. In addition, news headlines on this page may change during the day. Without delta optimization technology, a user must download the entire 70 KB with each subsequent visit to the page, even though the changes in the underlying HTML comprise only about 2 KB of the page.

The delta optimization technology allows the ACE to enable the content provider to dynamically calculate the content differences, or deltas, between subsequent content retrievals (on a per-user basis if desired) and send only those deltas for subsequent visits to the dynamic content. A user would retrieve 70 KB on the first visit to the home page but would need to retrieve only the 2 KB content delta on subsequent visits. With delta optimization, only the deltas between the subsequently requested pages are sent to the users. These deltas, which are encoded by using dynamic HTML, enable the ACE to directly update the client's browser cache, much like an origin server updates a traditional edge cache.

Figure 1-2 shows how the ACE fits into the network topology.

Figure 1-2 Simplified ACE Topology



In the figure, the ACE is located in the path of the content delivered from a content origination server (web server) to the browser but is situated close to the server. The ACE observes and modifies the content that flows through it to achieve bandwidth savings and increase user download performance.

The following topics describe ACE support for web browsers, server, and content for delta optimization:

- Web Browser Support
- Web Server Support
- Web Content Support
- Configurable Delta Optimization Modes

Web Browser Support

The ACE supports traffic directed to and from all web browsers that use the HTTP 1.0 and HTTP 1.1 protocols. HTTP 1.1 supports chunking by using the "Transfer-Encoding: Chunked" HTTP request header as described in section 14.40 of RFC 2068, "Hypertext Transfer Protocol -- HTTP/1.1."

The ACE supports full optimization for all major browsers on Windows 98, Windows NT 4, Windows 2000, Windows ME, and Windows XP as long as cookies and JavaScript are enabled in these versions:

- Microsoft Internet Explorer versions 5.5, 6.0, and later

- Netscape Communicator versions 4.8, 6.2, and later (also for Solaris 2.8, 2.9 and Redhat AS 3.0)
- AOL browser versions 6.0 and later
- FireFox 2.0 (also for Solaris 2.9)

All other browsers and configurations are supported without delta optimization.

When the user requests a page, the ACE identifies the user's browser type and applies those optimizations supported by the browser.

No configuration changes or software installations are required in the browser because the ACE can automatically detect the browser brand, version, and platform, cookie setting, and JavaScript setting.

Cookie and JavaScript support are detected as follows.

1. On the user's first visit to a site, the ACE inserts a JavaScript probe code in the page delivered to the user. This JavaScript code creates an ACE cookie on the user's system.
   - If the browser does not support JavaScript, JavaScript is disabled, or if the cookies are disabled, then the cookie creation fails.
   - Otherwise, the ACE cookie is created successfully.
2. When the user visits the site the second time, the presence of the cookie indicates that the browser supports JavaScript and cookies which signals to the ACE to deliver delta optimized content to the user.

Note The ACE also supports a user who subsequently reconfigures the browser to disable JavaScript. For more information, see the "Cookie Usage" section.

Web Server Support

The ACE supports all web servers and web application servers that support the HTTP 1.0 or 1.1 protocols on all platforms. HTTP 1.1 supports HTTP 1.1 chunked transfer encoding through the Transfer-Coding:Chunked HTTP request header as described in section 14.40 of RFC 2068, "Hypertext Transfer Protocol -- HTTP/1.1.".

Web Content Support

The ACE supports all web content without modification to the content or the server software. However, not all web content is suitable for delta optimization. Uncondensable content is passed through unmodified.

The following sections describe the limitations to delta optimization support:

- Character Set Support
- Unsupported HTML Elements

Character Set Support

Delta optimization occurs only on the HTML content that uses the ISO-8859-1 character set because the ACE supports single-byte characters only. All other types of content can pass through uncondensed.

The ACE detects the character set type by examining the Content-Type HTTP response header, which appears as follows: Content-Type: text/html; charset=iso-8859-1. If no character set specification is present, the ACE assumes it to be ISO-8859-1.

Unsupported HTML Elements

Certain types of HTML content are not condensable. Inline frames IFRAME tags), external scripts that use the include action (SCRIPT tags including the SRC attribute which allows authors to reuse code), and embedded objects OBJECT tags) cannot be delta optimized. While pages that contain these elements are delta optimized, the elements within the page pass through uncondensed.

All JavaScript code embedded in a page is passed through uncondensed and in the exact order in which it appears in the origin page. This step avoids problems related to JavaScript dependencies and execution order.

To configure the delta optimization operating parameters on the ACE to define the cacheable objects that should not be delta optimized, use the **delta** command in parameter map optimization configuration mode. For details, see Chapter 3, Configuring an Optimization HTTP Parameter Map.

Configurable Delta Optimization Modes

You use the delta optimization mode to specify whether the web pages to be delta optimized are common to all users or personalized for individual users. Your choice of delta optimization mode determines what kind of page deltas are generated by the ACE.

The ACE supports two delta optimization modes:

- All-user mode
- Per-user mode

In the all-user mode, the delta is generated against a single base file that is shared by all users of the URL. The all-user delta optimization mode is usable in most cases—even in the case of dynamic personalized content, if the structure of a page is common across users. The disk space overhead is minimal (the disk space requirements are determined by the number of condensed pages, not the number of users).

In the per-user mode, when a specific user requests a URL, the delta for the response is generated against a base file that is created specifically for that user. The per-user delta optimization mode is useful in situations where the contents of a page (including layout elements) are different for each user. It delivers the highest level of delta optimization. However, a copy of the base page that is delivered to each user has to be kept in the ACE cache, which increases the requirements on the disk space for the ACE cache. The per-user delta optimization mode is useful for content privacy because base pages are not shared among users.

To control the delta optimization mode used by the ACE, use the **delta** command in parameter map optimization configuration mode (see Chapter 3, Configuring an Optimization HTTP Parameter Map).

Adaptive Dynamic Caching

Adaptive dynamic caching enables the ACE to fulfill requests for dynamic or personalized information, offloading application servers and databases. Adaptive dynamic caching significantly improves application response time, reduces the server load, and enables more concurrent users to be served, resulting in improved scalability and lower ongoing server upgrade costs. The performance assurance

caching policy enables the ACE to monitor the server load in real time and make intelligent closed-loop content expiration decisions to maximize site performance and hardware resources during peak traffic load.

Adaptive dynamic caching enables the ACE to cache the dynamic content such as the content created by an active server page (ASP) script or application server. The dynamic content is not usually cached, because the generated content may contain up-to-the-minute data; however, by allowing flexible configuration and algorithmic processes, the ACE allows you to cache this kind of content for some period of time.

Adaptive dynamic caching includes the following features:

- Cache parameterization—Differentiates a response by more than the URL and its query parameters. It allows other parameters such as cookie values, HTTP header values, and the HTTP method used. This feature allows the ACE to cache multiple responses for a single URL, depending on the specified cache parameters. Cache parameterization applies to both static (FlashForward) and dynamic caching.
- Expanded expiration rules—Sets the automatic expiration of the cached content based on the time or through performance assurance with load-based expiration.
- Delta Cache—Stores the delta content in a cache (a memory-only object) when the original HTML content is in the dynamic cache and a rebase has not occurred.

Adaptive dynamic caching does not have any default configuration that suits a wide variety of situations automatically. You must configure this feature specifically for a website or application. You can configure adaptive dynamic caching in an HTTP optimization parameter map, and then associate it with an optimization HTTP action list, which allows different caching to be applied to different sets of URLs specified in a Layer 7 server load-balancing policy.

For guidelines on when to use dynamic caching and the steps required to implement and configure it, see the "Dynamic Caching Configuration" section.

Dynamic Caching Configuration

Before deploying the ACE, we recommend that you analyze traffic patterns to determine the suitability of the various features offered by the ACE (such as FlashForward and adaptive dynamic caching). There are many ways to determine the optimal configuration settings, however, you should follow these guidelines for configuring dynamic caching.

- Dynamic Caching Guidelines
- Configuring Dynamic Caching

For details about enabling the cache optimization for the corresponding URLs, see Chapter 2, Configuring an Optimization HTTP Action List.

For details about configuring dynamic caching in a parameter map, see Chapter 3, Configuring an Optimization HTTP Parameter Map.

Dynamic Caching Guidelines

Use dynamic caching in the following situations:

- Dynamic caching is useful only if the response can be reused over some period of time, even if you have a small period of time.
- Dynamic caching is useful only if a cached response can be used multiple times before the response expires, as follows:
  - The same response is usable by multiple users within a certain period of time; that is, the content is sharable. This feature is useful even if the cache expiration period is very small if enough users access the response within that period. An example would be a hierarchical directory browser application (for example, a catalog store or a document management application) where the directory is generated dynamically but is the same view seen by all users. Dynamic caching can avoid the cost of regenerating the directory pages for each user.
  - The response is usable by only a single user (it is personalized), but the user accesses it multiple times. The reuse of the response occurs across multiple browser sessions for that user. For example, the response generated in one session can be used unchanged in a second session, or the reuse of the response can occur only within a single browser session because the response is tied to that particular session.

- Dynamically cacheable responses are identifiable using one or more of the following:
  - The URL matches a regular expression.
  - Specific query parameters are present.
  - Specific cookies in the request are present.
  - Specific HTTP headers in the request are present.

  For example, a particular application uses the same URL for various actions, only some of which are cacheable. Consider the case where the URLs in the application are:

  1. http://xyz.com/doit.jsp?action=login&username=xyz

  2. http://xyz.com/doit.jsp?action=browse&level=1

  3. http://xyz.com/doit.jsp?action=browser&level=2

  All the URLs are the same, however, the response to (1) cannot be cached, but assume the response to (2) and (3) can be cached.

Do not use dynamic caching in the following situations:

- The response becomes stale immediately upon delivery. Examples of this are as follows:
  - The response sets cookies specific to that session. For example, the response to a login page is specific to a particular session.
  - The response contains data specific to a previous action in the session. For example, a confirmation number for a transaction that was just executed is not cacheable.
- The life of a response is indeterminate; that is, the response contains data that becomes stale based on a future action. For example, the portfolio page of a brokerage account user changes when the user executes transactions.
- Different versions of the response cannot be distinguished by using the URL, the URL query parameters, or the cookies in the request. For example, the response contains some personalized settings, such as the name of the user; however, no URL query parameter or cookie directly identifies the user. The request must identify the user in order for the origin server to generate the personalized response (for example, a session cookie). You could dynamically cache a version of the page that is different for each user session. However, in this case, dynamic caching is useful only if the user accesses the page multiple times within a single browser session.

Configuring Dynamic Caching

Once the appropriateness of dynamic caching has been established for a page, follow the steps outlined in this section to properly configure dynamic caching in the ACE.

To configure dynamic caching in the ACE, follow these steps:

Step 1 Identify the URL or URLs by developing the regular expression to match the URL(s) in a Layer 7 server load-balancing class map. Performing this step should help you form the short list of URLs to target as follows:

- Run the performance test using Accelerometer or another performance testing program on the LAN. The LAN is used so that the focus is on identifying server latency problems, not network latency problems.
- Examine the error log to look at entries where the origin server response time is large. Large means that it is a significant part of the page download time as measured by the end user. For example, if a page download on the LAN takes 10 seconds, then you should examine any individual component download with an origin server response time of greater than a second.

Step 2 Identify the dynamically cacheable instances of the URL by performing regular expression matching the URL. If the regular expression matches many responses, some of which can be dynamically cached and some which cannot be, determine if the cacheable responses can be separated from the uncacheable ones. Test for the presence or absence of specific query parameters, cookies, or HTTP headers.

Step 3 Identify the versions of the response. Once the dynamically cacheable instances are identified, then the next step is to figure out how many different versions of the response are to be cached by using cache parameterization. Look at the request parameters and the response content to determine what defines the content of the response. Examine the following request parameters:

- URL and its components
- Query parameters
- Cookies
- HTTP header values

Step 4 Develop a minimal set of parameters that uniquely determines the content of the response. Based on this response, modify the cache key by defining the **cache key-modifier** and **cache parameter** commands in an optimization parameter map for this URL (see Chapter 3, Configuring an Optimization HTTP Parameter Map) as follows:

- The response is determined uniquely by the URL and all its parameters. In this case, nothing needs to be done because it is the default cache key.
- The response is determined by only some of the parameters. In this case, use the **cache parameter** command to specify those parameters. For example, a site may have the following URL:

  ```
  http://xyz.com/dosomething.asp?action=browse&dir=x&session=13345.
  ```

  The contents of the response may be determined by the **action** and **dir** query parameters. The **session** query parameter does not affect the response. You could specify the following **cache parameter** commands:

  ```
  host1/Admin(config-parammap-optmz)# cache parameter
  $http_query_param(action)
  host1/Admin(config-parammap-optmz)# cache parameter
  $http_query_param(dir)
  ```

- The response is determined by the value of some cookie(s). You can add them to the optimization parameter map as follows:

  ```
  host1/Admin(config-parammap-optmz)# cache parameter
  $http_cookie(foo)
  ```

- Some parts of the URL need to be ignored. For example, some sites embed the session ID in the URL itself, but the response is not dependent on the session ID. You can use URL subexpressions in the **cache key-modifier** command in an optimization parameter map. For example, assume that a site has a URL of the form:

  ```
  http://xyz.com/sess12345/dosomething.asp?action=browse&dir=x.
  ```

  If sess12345 is not affected by the response, then specify the URL-matching regular expression (in the Layer 7 SLB class map) and the **cache key-modifier** command as follows to eliminate the (sess*NNNNN*) string from the cache key:

  ```
  host1/Admin(config)# class-map type http loadbalance match-any
  Example_Classmap
  host1/Admin(config-cmap-http-lb)# match http url
   (.*)/(sess.*)/(dosomething.asp)
  ```

```
host1/Admin(config-cmap-http-lb)# exit
host1/Admin(config)# parameter-map type optimization http
OPTIMIZE_PARAM_MAP1
host1/Admin(config-parammap-optmz)# cache key-modifier $(1)/ $(3)
```

Step 5 Identify how long the response can be cached by considering the content. For example, a new item may be viewed for 3 hours, after which it becomes stale.

The **cache ttl** timing related optimization parameter map command configures the minimum and maximum time-to-live selection in seconds as follows:

- Minimum time-to-live—Specifies the minimum time that the content can be cached, which corresponds to the lifetime of the content. For a new item that is valid for 3 hours, this value would be 3 x 60 x 60 = 10800 seconds.
- Maximum time-to-live—Determines how the ACE handles the situation when the object has passed its minimum time-to-live value.

You can also configure the object expiration by using performance assurance with load-based expiration, which allows you to dynamically increase the time-to-live setting of cached responses when the current response time (the average computed over a short time window) from the origin servers is larger than the average response time (the average computed over a longer time window) by a threshold amount. Similarly, the TTL is dynamically decreased if the reverse situation is true.

For details about configuring cache expiration, see Chapter 3, Configuring an Optimization HTTP Parameter Map.

FlashForward Object Acceleration

FlashForward object acceleration extends the ACE appliance's bandwidth usage reduction and download acceleration benefits to objects that are embedded within HTML pages. This feature combines the local object storage with dynamic renaming of embedded objects to enforce object freshness within the parent HTML page.

FlashForward renames and caches static objects in the ACE, alters embedded object HTTP headers to extend their validity within the browser cache, and modifies the URL references in the parent HTML code to refer to the renamed objects so that the client requests only objects known to be new or modified at the

time of the HTML request. This technique results in significantly accelerated page downloads by the client and reduced upstream traffic that is associated with object validation requests.

FlashForward eliminates the network delays associated with embedded web objects such as images, style sheets, and JavaScript files. Without the ACE, the user experiences delays when pages with graphic images load because each object requires validation to ensure that the user has the latest version. Object validation can result in 20 KB or more of unnecessary upstream traffic. Each validation involves an HTTP request from the client to the server. FlashForward enforces embedded object version management at the server. All object validity information is carried in the single download of the parent HTML document, which eliminates unnecessary validation requests.

The web's current object freshness validation mechanism forces the client to assume that all objects cached within the browser are invalid (stale) in subsequent sessions until the server explicitly communicates its object validity to the client. This approach can create significant page load delays on subsequent visits to a previously cached page because it forces the client to issue a validation request for each object. A page load delay can be quite lengthy for pages that embed multiple objects because the objects cannot be rendered until the client-to-server round-trips are completed. In addition, this approach may waste significant upstream bandwidth.

FlashForward places the responsibility for validating object freshness on the ACE rather than on the client, reversing the process and making it more efficient. FlashForward guarantees that clients request only the latest objects and never issue validation requests for objects in the browser cache that the ACE has determined to be valid. Working together with delta optimization (see the "Optimization Traffic Policies and Typical Configuration Flow" section), small delta pages are used to deliver the information that references new objects. With FlashForward, the client never needs to validate the freshness of browser-cached objects with the origin server, which significantly accelerates page downloads and reduces both upstream and downstream traffic that is associated with object validation requests.

For details about how FlashForward operates, see the "FlashForward Operation" section.

To configure FlashForward, use the **flashforward** or **flashforward-object** command in action list optimization mode (see Chapter 2, Configuring an Optimization HTTP Action List).

Using Application Acceleration FlashForward With IP Address Stickiness

When you operate the ACE with application acceleration, if your configuration includes the following elements:

- FlashForward object acceleration to extend the ACE appliance's bandwidth usage reduction and download acceleration benefits to objects that are embedded within HTML pages.
- IP address stickiness to stick a client to the same server for multiple subsequent connections as needed to complete a transaction.

you may find that there will more than one sticky entry created for the same session when a single user (one client IP address) attempts to retrieve one HTML page. The creation of multiple sticky entries is the expected behavior; in addition to the sticky entry created for the user session, a second sticky entry will be created for the application acceleration cache renew requests for the FlashForward embedded objects.

This behavior can be encountered when one of the following conditions occur:

- The client already contains the cached page in the browser cache.
- The application acceleration cache expires maximum setting (the **expires-setting** command in parameter map optimization configuration mode) is at the default of 300 seconds.
- The IP address sticky timeout setting (**timeout** command in sticky-IP configuration mode) is set to a lower value (for example, two minutes).

Under these conditions, when the client request an HTML page, the request is then load-balanced to multiple real servers. With IP address stickiness enabled, you may expect that the request would stick with a single real server and serve the HTML page from that server.

For example, in the **show serverfarm** output illustrated below, both servers increment and form two sticky entries.

```
switch/Admin# clear serverfarm WWWfarm
switch/Admin# show serverfarm WWWfarm
 serverfarm     : WWWfarm, type: HOST
 total rservers : 2gg
 active rservers: 2
 description    : -
 state          : ACTIVE
```

```
 predictor      : ROUNDROBIN
 failaction     : -
 back-inservice    : 0
 partial-threshold : 0
 num times failover       : 0
 num times back inservice : 0
 total conn-dropcount : 0

 ---------------------------------

----------connections-----------
       real                  weight state        current    total
failures

---+--------------------+------+-----------+----------+----------+--
-------
   rserver: WWWserver
       192.168.10.12:0       8      OPERATIONAL  1          1
0
         max-conns            : -         , out-of-rotation count : -
         min-conns            : -
         conn-rate-limit      : -         , out-of-rotation count : -
         bandwidth-rate-limit : -         , out-of-rotation count : -
         retcode out-of-rotation count : -
         load value           : 0

   rserver: WWWserver2
       192.168.10.99:0       8      OPERATIONAL  1          1
0
         max-conns            : -         , out-of-rotation count : -
         min-conns            : -
         conn-rate-limit      : -         , out-of-rotation count : -
         bandwidth-rate-limit : -         , out-of-rotation count : -
         retcode out-of-rotation count : -
         load value           : 0
```

Just-In-Time Object Acceleration

Just-in-time object acceleration enables acceleration of noncacheable embedded objects, resulting in improved application response time. This feature eliminates the need for users to download these objects on each request. Instead, the ACE automatically tracks the freshness of each object in real time. If a requested object has not changed, the ACE instructs the client to use its cached version of the

object. If an object has changed, the ACE delivers it to the client. The ACE delivers the object only if it determines that the object has changed, guaranteeing the optimal application response times for all users.

The static content, such as images, is handled by the ACE with FlashForward, and dynamic HTML is handled by the ACE with delta optimization. Just-in-time object acceleration is useful for dynamic content that cannot be handled by delta optimization, such as under the following conditions:

- The HTML content is dynamic and larger than the maximum condensable page size (250 KB).
- The content is marked by the origin server as expired or not cacheable.

We recommend that you use just-in-time object acceleration with HTTP compression (see the *Cisco 4700 Series Application Control Engine Appliance Server Load-Balancing Configuration Guide)*.

Just-in-time object acceleration functions as follows:

1. The browser requests an object and the ACE fetches it on behalf of the browser request.
2. The ACE constructs and inserts an ETag (entity tag) header by using an MD5 hash of the content, and then it sends the object to the browser. The ETag (entity tag) is a request header that you can use to identify different versions of the same object.
3. All subsequent requests from the browser include this ETag header, which the ACE compares with a recomputed MD5 hash of the latest origin server content as follows:
   - If the content has not changed, then the ACE returns a 304 (Not Modified) response and no data.
   - If the content has changed, the ACE retrieves the new content and resets the ETag.

To configure just-in-time object acceleration, use the **dynamic etag** command in action list optimization mode (see Chapter 2, Configuring an Optimization HTTP Action List).

When you operate the ACE with application acceleration, and you define a policy map that includes dynamic ETag (entity tag) and HTTP compression, you may encounter behavioral issues when using a Microsoft Internet Explorer web browser (typically, Internet Explorer 6.0) that prevents the dynamic ETag function from properly leveraging the web browser cache. With HTTP compression and

dynamic ETag configured, upon an Internet Explorer web browser refresh of the same page without any content changes, the browser will continue to retrieve the new (unchanged) content along with a new ETag header and the HTTP “200 OK” request succeeded response. Typically, when you use dynamic ETags and the web content has not changed, the browser receives a 304 Not Modified response and no data. This response prevents users from downloading objects on each request.

This operating consideration with the Microsoft Internet Explorer web browser does not result in broken pages or functionality but rather in a failed attempt at application acceleration, which results in the unnecessary download of objects that should be cached.

This behavior does not occur with the Firefox 2.0 web browser, so we recommend that you use the Firefox 2.0 web browser instead of the Internet Explorer 6.0 web browser if you plan to configure dynamic ETag (entity tag) with HTTP compression.

Base File Management

The ACE uses a caching mechanism to optimize performance. It implements an automatic, transparent cache for base pages, where the least-used pages are discarded from the cache when it becomes full.

Base file selection policy is similar to the canonical URL feature (see the “Canonical URL” section) but provides more flexibility in how you specify a base file to be shared among a group of URLs. The base file selection policy allows you to identify regular expressions that define how URLs should be generalized. For example, Amazon.com uses URLs that look like the following:

http://www.amazon.com/exec/obidos/tg/browse/-/289728/ref=k_kh_ln_bp_2/105-3394538-7390300

http://www.amazon.com/exec/obidos/tg/browse/-/289737/105-3394538-7390300

These URLs are parameterized by ID numbers within the path, rather than the typical question mark (?) character. You can configure the base file selection policy to determine that the common URL in this example is http://www.amazon.com/exec/obidos/tg/browse/-/, and that the base file should be created for this base URL. All similar requests would share this same base file.

Anonymous Base Files

The ACE incorporates an anonymous base file feature to address user privacy concerns. This feature, which is an all-user delta optimization mode option (see the "Configurable Delta Optimization Modes" section), enables customers to use the ACE nodes to deliver personalized confidential content such as online trading accounts, banking statements, business accounts, and so on. Typically, customers use this feature with SSL to enable secure and private condensed content delivery.

Information that is common to a large set of users is generally nonconfidential or nonuser specific. Conversely, information that is unique to a specific user or common across a very small set of users is generally confidential and/or user specific. This feature enables the ACE to create and deliver base files that contain only information that is common to a large set of users. No information unique to a particular user (or across a very small subset of users) is included in anonymous base files. Anonymous base files eliminates any context for the data within a base file, making it impossible to associate the information with a given user. The anonymous base file does not represent any initial (potentially confidential) content as viewed by the first visitor to a particular URL.

As an example, consider two numbers **m** and **n**, where **m** represents the anonymity level and **n** represents the base file sample size. The anonymous base file feature seeks to create a shared base file that contains the content that is common only to **m** out of **n** base files (and users). For example, if **m**=4 and **n**=20, the anonymous base file will contain the content that is common only to at least 4 of 20 user-specific base files. Any content that is unique to any one of the 20 base files will not be included in the anonymous base file. In addition, the content that is common to fewer than four base files will not be included. These 20 base files will be of a per-user type created only to enable this feature (these per-user base files will not be used to condense the content). The base files in the base file sample size are chosen as the first **n** unique requests from the unique browsers for the given URL.

You can configure the anonymity level (**m**), where **m**=0 (no anonymity will be enabled). The ACE will then select **n**=max(5,3**m**) to ensure highly anonymous base files. That is, **n** is set to the larger of 5 or 3 x **m**.

You can configure the anonymous basefile feature by using the following parameter map optimization mode commands:

- **basefile anonymous-level—**Specifies an additional base file anonymous level that corresponds to **m**.

- **delta all-user**—Specifies the all-user delta optimization mode

These commands are described in Chapter 3, Configuring an Optimization HTTP Parameter Map.

Class-Based Condensation

Class-based condensation allows a common base file to be shared among multiple URLs, known as a class of URLs. This mode is different from the normal URL-based mode in which a separate base file is maintained for each URL that is condensed.

Class-based condensation allows you to define classes of web pages that have similar layout and/or content. For a particular class of pages, any page within the class is condensed against a single master base page that represents all pages in the class. One document can be condensed against a similar, previously retrieved document rather than being condensed against a previously downloaded version of the same document, as in URL-based condensation.

A class of URLs is defined by specifying a regular expression that matches all URLs in the class. For example, the expression http://host/thisdir/* groups all files in the specified path into one class. If this path contained the two files http://host/thisdir/first.html and http://host/thisdir/second.html, they would share a common base file.

Canonical URL

The ACE uses the canonical URL function to modify a parameterized request to eliminate the question mark (?) and the characters that follow to identify the general part of the URL. This general URL is then used to create the base file. The ACE uses the canonical URL to map multiple parameterized URLs to a single canonical URL.

Use the **canonical-url** command in parameter map optimize mode to specify a base file selection by identifying a string that contains a canonical URL regular expression. This string is a regular expression that is used to match a variety of actual URLs. All matched URLs share a single base file. The **canonical-url**

command can contain parameter expander functions that evaluate to strings. See Chapter 3, Configuring an Optimization HTTP Parameter Map for details. Table 3-3 lists the parameter expander functions that you can use.

For example, the following two URLs would both be reduced to the URL http://www.servers.com/books:

http://www.servers.com/books?id=235

http://www.servers.com/books?id=576

As a result, both parameterized URLs would share the same base file that represents the canonical URL http://www.servers.com/books. Condensation levels will be relatively low if these original URLs reference two pages that do not share much content or layout (relatively large delta files could be delivered for requests across parameterized URLs that do not share the content or layout).

An example of using the canonical URL is as follows. Assume that a catalog website has the following pages:

http://server/catalog/business?category=pencils

http://server /catalog/business?category=erasers

http://server /catalog/consumer?category=pencils

http://server /catalog/consumer?category=erasers

In this example, assume that the pencils pages for both the business and consumer sections have a lot of common content and the erasers pages have similar content. It would be efficient if you could share the base-pages for these categories. That is, the base-page for the following two pages would be the same as follows:

http://server/catalog/business?category=pencils

http://server/catalog/consumer?category=pencils

When you use the **canonical-url** command, the ACE can match all four URLs:

```
host1/Admin(config-parammap-optmz)# canonical-url $(1)/
$http_query_param(category)
```

For example, if the request URL is

http://server/catalog/business?category=pencils

then (1) is the http://server/catalog string and http_query_param(category) expands to the pencils string. The resulting canonical URL is

http://server/catalog/pencils

The same is true for the following URL:

http://server/catalog/consumers?category=pencils.

In this situation, the two URLs will share the same base file.

Similarly, the canonical URL for the eraser URLs will be as follows:

http://server/catalog/erasers

and they will share the same base file.

Smart Rebasing

Smart Rebasing updates the base file that is used for generating deltas. Because the base content of a site often changes over a period of time, the size of the generated deltas can grow relatively large. To maintain the effectiveness of the delta optimization process, the base files are automatically updated as required.

Smart Rebasing enables the ACE to instantly rebase URLs when appropriate and to maintain a copy of the old base page so that subsequent requests for it can be fulfilled.

The **rebase flashforward-percent** command in parameter map optimization mode provides a threshold control that allows you to rebase based on the percentage of FlashForward URLs in the response. Where you used the **rebase delta-percent** command in parameter map optimization mode to trigger rebasing when the delta response size exceeds the threshold as a percentage of the base file size, you use the **rebase flashforward-percent** command to trigger rebasing when the difference between the percentages of the FlashForward URLs in the delta response and the base file exceed the threshold.

With Smart Rebasing, the ACE tracks the number of delta responses sampled, how many of the responses have a delta size bigger than the **rebase delta-percent** command threshold, and how many of the responses have too many FlashForward URLs. When a reasonable sample size (for example, 10) is reached, the ACE looks at the percentages of delta responses that have exceeded the configured rebase delta-percent and rebase FlashForward percentage thresholds. By default, rebasing is automatically triggered when either percentage exceeds 50 percent. Smart rebasing enables the ACE to automatically rebase a page when it determines that the existing base file does not result in minimally sized delta responses for the majority of requests.

Smart Rebasing improves the overall ACE performance and content acceleration because it ensures that the delta optimization occurs at all times, even when a rebase occurs.

MIME-Type Exclusion

Because some content providers label the text/HTML content as nontext/HTML Multipurpose Internet Mail Extension (MIME)-type messages in the HTTP entity header field to prevent web crawlers, the MIME-type exclusion allows you to explicitly configure MIME types as uncondensable. For example, assume that you configure the web server to report JavaScript as MIME type application/x-text. If you configure this feature to identify this MIME type as uncondensable, the ACE does not condense responses for this MIME type. When you disable this feature, the ACE condenses responses for this MIME type.

To list the MIME types that are not to be delta optimized in the mimetypes.conf file, use the **delta exclude mime-type** command in parameter map optimization mode. If this file is empty, or does not exist, then all MIME types are considered condensable and compressible.

For details about configuring MIME-type exclusion, see the Chapter 3, Configuring an Optimization HTTP Parameter Map section.

Cookie Usage

To determine the browser support for cookies and JavaScript, the ACE inserts JavaScript code into the page returned to a user on the user's first visit to a site. When executed by the client browser, this code creates a cookie with a randomly generated 128-bit user ID. Client browsers that do not support JavaScript will ignore the script, and no ACE cookie will be created.

When the client makes a second request, the ACE looks for the ACE cookie to verify the client JavaScript and cookie support. If the ACE sees the ACE cookie, it assumes that the client supports JavaScript. If it does not see the ACE cookie, it assumes that the client does not support JavaScript or cookies (or that this may be the first request from this client), and it does not provide delta optimization for this user's requests.

The ACE cookie has the attributes listed in Table 1-1.

Table 1-1 Cookie Properties

Attribute	Value
Name	ACCCDN
Life	30 days
Domain	Same as the target site
Path	"/" (the entire site)
Value	128-bit randomly generated user ID. This ID uniquely identifies the user to the ACE.

The ACE supports automatic cookie expiration, which enables the ACE to handle a potential failure of the user by disabling JavaScript after an initial JavaScript-enabled visit. Without this support, the browser would display a blank page when it retrieves the condensed content (which is wrapped in JavaScript) from the ACE. To handle this situation, the ACE includes a <NOSCRIPT> tag in its responses to clients. If JavaScript is disabled on the browser after the initial JavaScript-enabled request, the client execution of this HTML code automatically forces the client to expire the ACE cookie and fetch subsequent pages uncondensed (until JavaScript is reenabled).

If JavaScript is disabled, the client executes the HTML code within the <NOSCRIPT> tags to fetch a URL such as http://www.example.com/?cisco=removeCookie. When the ACE sees this request, it issues an HTTP 302 Temporary Redirect to the client, redirecting it to the originally requested page. The response also includes a Set-Cookie HTTP header that immediately expires the client's ACE cookie.

AppScope Performance Monitoring Using the Cisco AVS 3180A Management Station

The optional Cisco AVS 3180A Management Station runs the Management Console that includes a series of database, management, and reporting features, including AppScope reporting, for the ACE optimization functionality.

The AppScope Performance Monitor provides a browser-based reporting facility that enables enterprises to efficiently track application performance. AppScope's reporting engine provides detailed graphical performance monitoring results with drilldown reports.

All AppScope Performance Monitor data is stored in a self-contained relational Postgres SQL database, which allows an organization to use its own reporting tools, such as Crystal Reports, or to create its own custom performance monitoring reports.

For details about using AppScope performance monitoring and generating reports, see Appendix A, Using the Optional Cisco AVS 3180A Management Station for Reporting.

Detailed ACE-Client Interaction

This section describes the interaction of the ACE and a client web browser over a series of two visits that the individual client makes to a web page.

Note The examples in this section cover the all-user delta optimization method.

This section contains the following topics:

Visit One—New Client

The process begins with the client's first visit to the web page since the ACE has been deployed.

In this visit, the ACE acts as a transparent proxy for the origin server, simply returning the web page requested by the client. If the client is one that is known to support JavaScript, the ACE also embeds some JavaScript code into the returned page (for details, see the "Visit One JavaScript Example" section). JavaScript attempts to create an ACE cookie on the client's system. If JavaScript is successful, the ACE knows that the client actually does support JavaScript and

that the function is enabled. The presence of this cookie instructs the ACE that when this client requests this page (or a page from this class) in the future, the ACE can return condensed content.

Visit One JavaScript Example

This example shows a page that was sent to a client that is visiting www.foo.com for the first time. The JavaScript code that installs the ACE cookie is shown at the top. The ACE adds this JavaScript to the existing HTML page content.

```
<SCRIPT LANGUAGE="JavaScript">
//<!--
document.cookie="ACCCDN=5.0-f98f1ec8-7b57-402f-b23b-77f708a9a26b;
path=/;expires=Sat, 16 Jun 2007 18:50:05 GMT";
//-->
</SCRIPT>
<html><head><title>Foo!</title><base href=http://www.foo.com/>
<meta http-equiv="PICS-Label"
content='(PICS-1.1 "http://www.rsac.org/ratingsv01.html"
l gen true for "http://www.foo.com" r (n 0 s 0 v 0 l 0))'>
</head><body><center><form action=http://search.foo.com/bin/search>
<map name=m><area coords="11,0,73,52" href=r/a1>
<area coords="74,0,142,52" href=r/p1>
<area coords="143,0,212,52" href=r/m1>
<area coords="462,0,531,52" href=r/wn>
<area coords="532,0,600,52" href=r/i1>
<area coords="601,0,665,52" href=r/hw>
</map><img width=674 height=53 border=0 usemap="#m"
src=http://us.a1.yimg.com/us.yimg.com/i/ww/m5v4.gif alt=Foo><br>
<table border=0 cellspacing=0 cellpadding=3 width=640><tr><td
align=center width=205>
    etc...
</body></html>
```

Visit Two—Client Returns

The process continues with the client's next visit to the web page. The example in this section represents the second and all subsequent times that a client returns to the web page after the first visit.

In this visit, the ACE determines if the client supports JavaScript by checking for the presence of the ACE cookie that was created during the first visit. If the ACE finds the cookie, it knows that JavaScript is enabled and the client can handle delta pages. The ACE then generates and delivers a delta page to the client. For details about the delta page contents, see the "Visit Two JavaScript Example" section.

Note The first delta page returned by the ACE requires two requests from the client to the ACE because the base page is also delivered. This special base page contains JavaScript that is used to apply the deltas on subsequent visits. This base page is also stored by the ACE, so it can be used to calculate future deltas. On subsequent visits, only the small delta pages are delivered, unless rebasing is required.

On each subsequent visit, the ACE always checks for the presence of the ACE cookie. If the ACE cookie is not present, the ACE treats the interaction as a first visit and attempts to create a new cookie as usual.

If a base page file referenced from a delta page is no longer available to the browser—that is, it is no longer in the browser cache, not available on web caches, and not available on the ACE—the delta file returned by the ACE is not useful. Without the base page, the browser would not be able to access the content delivered by the ACE because the client would attempt to apply ACE deltas to a nonexistent base page. To handle this situation, the ACE includes JavaScript code in responses to clients that forces the browser to reload the base page and bypass the cache if the base page is unavailable.

Visit Two JavaScript Example

This example shows a page that was sent to a client that is visiting www.foo.com for the second time:

```
<script type="text/javascript">var isACCBaseCorrect=false;</script>
<script type="text/javascript"
src="/4grv3hs41d2bkt4wj41kcotmlh/986848530/ausr/_acc_http_//www.foo.co
m/">
</script>
<noscript><META HTTP-EQUIV="Refresh" CONTENT="0;
URL=http://www.foo.com/?ciscoacc=removeCookie">
Please click on <a href="http://www.foo.com/?ciscoacc=removeCookie">
this url</a> if the page is not refreshed automatically in a few
seconds
</noscript>
<script type="text/javascript">
```

```
if (!isACCBaseCorrect)
    document.location.reload(true);
</script>
<script type="text/javascript" >
/*
Portions Copyright (c) 2005-2007 by Cisco Systems, Inc. All rights
reserved.
SendDelta
*/
document.open('text/html', 'replace');
fgn_b(0, 861);
fgn_o('84022.657463.2834087');
fgn_b(881, 31);
fgn_o('528320');
fgn_o('art2');
.
.
.
fgn_o(' - Shop until midnight, Dec. 20');
fgn_b(7539, 11017);
fgn_flush();
document.close();

fgn_flush();
</script>
```

The following discussion breaks down each section of the file in detail:

```
<script type="text/javascript">var isACCBaseCorrect=false;</script>
```

This segment of code defines the default ACCBaseCorrect variable to be false. This variable will be set to true when a condensed page is successfully handled. This variable is used to enable the ACE's base file recovery that allows it to handle a potential failure scenario where a base file referenced within a delta page is no longer available to the browser (it is no longer in the browser cache, not available on web caches, and not available on the ACE node). Without this feature, the browser cannot access the content delivered by the ACE because the client would attempt to apply ACE deltas to a nonexistent base page. This feature addresses this problem through JavaScript code within the ACE response that forces the browser to reload the page (and bypass the cache) whenever the base page is irretrievable and fetch a new base page from the ACE.

```
<script type="text/javascript"
src="/4grv3hs41d2bkt4wj41kcotmlh/986848530/ausr/_fgn_http_//www.foo.co
m/">
</script>
```

This segment of code refers the client to the base file for the requested content. The base file can be retrieved from a network cache or the ACE if it is not available in the browser cache. This base file contains the original requested content and additional function definitions used by the client to construct subsequent pages from content deltas. The base filename is not the same as the originally requested page. As a result, base files are retrieved only by clients that use an ACE.

Examine the base file naming convention in this example. The first string, 4grv3hs41d2bkt4wj41kcotmlh, is the ACE ID that uniquely identifies the ACE that generated this base file. It is a one-way hash based on the MAC address, IP address, and port of the ACE node. The second string, 986848530, represents a modification time stamp of the base file that enables the ACE to detect the base file version changes.

```
<noscript><META HTTP-EQUIV="Refresh" CONTENT="0;
URL=http://www.foo.com/?ciscoacc=removeCookie">
Please click on <a href="http://www.foo.com/?ciscoacc=removeCookie">
this url</a> if the page is not refreshed automatically in a few
seconds
</noscript>
```

This segment of code enables the automatic cookie expiration feature, which allows the ACE to guarantee the content delivery in scenarios where JavaScript is disabled on the client. In this scenario, the client explicitly disables JavaScript support subsequent to an initial JavaScript-enabled visit. Without this feature, the browser would display a blank page when it retrieved the condensed content (JavaScript) from the ACE. The solution is to include this <NOSCRIPT> tag in the ACE responses to the clients. If JavaScript is disabled on the browser after the initial JavaScript-enabled request, the client execution of this code automatically forces the client to expire the ACE cookie and fetch subsequent pages uncondensed (without the ACE-generated JavaScript coding).

```
<script type="text/javascript">
if (!isACCBaseCorrect)
    document.location.reload(true);
</script>
```

This code enables the browser to reload the page uncondensed, bypassing the cache whenever the base file is unavailable. This failover mechanism ensures that the client can always retrieve the content even when the base files are unavailable.

```
<script type="text/javascript">
/*
```

```
Portions Copyright (c) 2005-2007 by Cisco Systems, Inc. All rights
reserved.
SendDelta
*/
document.open('text/html', 'replace');
fgn_b(0, 861);
fgn_o('84022.657463.2834087');
fgn_b(881, 31);
fgn_o('528320');
fgn_o('art2');
.
.
.
fgn_o(' - Shop until midnight, Dec. 20');
fgn_b(7539, 11017);
fgn_flush();
document.close();

fgn_flush();
```

This segment of code represents the content delta information. The simple string-manipulating JavaScript functions defined in the base file enables the client to construct the newly requested page from the previously retrieved base file.

Cache Control Headers

The delta pages that the ACE sends to the clients use the same HTTP cache control headers as the original page served by the origin server to allow the network caches to cache these pages in the same manner as the original page.

By default, the base pages that are sent to the clients by the ACE use cache control headers that enable caching for 30 days to leverage network edge caches.

FlashForward Operation

The following sections describe how the FlashForward function operates in the ACE:

- FlashForward Overview
- FlashForward and Delta Optimization Options
- FlashForward and Repeat Visits

- How FlashForward Works With Delta Optimization
- How FlashForward Works Without Delta Optimization
- How FlashForward Works With CDN URLs

FlashForward Overview

FlashForward object acceleration accelerates embedded image and object delivery by reducing both the upstream and downstream traffic that is associated with object validation requests.

An embedded object that contains a Last-Modified date and an Expires date indicates that the object has not expired in the browser cache. A visit in a new browser session to the HTML page that references the object does not trigger any kind of HTTP GET request for the object. FlashForward reduces the upstream HTTP request traffic and accelerates the page delivery.

FlashForward and Delta Optimization Options

FlashForward and delta optimization are independent mechanisms. While the recommended configuration is to use both delta optimization and FlashForward simultaneously for optimal acceleration and bandwidth savings, it is possible that you could configure FlashForward by itself or delta optimization by itself.

We recommend that you use both delta optimization and FlashForward for optimal results, as described in the "How FlashForward Works With Delta Optimization" section.

You might configure only FlashForward without delta optimization (see the "How FlashForward Works Without Delta Optimization" section) for sites with small HTML pages that contain many cacheable embedded objects. This type of configuration does not support accelerated HTML delivery in repeat visits. This configuration, however, does support accelerated embedded object delivery in repeat visits, typically across browser sessions. This type of configuration eliminates the requirement that client browsers must support cookies or JavaScript-based dynamic HTML (DHTML). FlashForward by itself supports all browsers automatically, but delta optimization requires a browser that can support DHTML.

You might configure delta optimization without FlashForward for sites with large HTML pages that contain few cacheable embedded objects. This type of configuration supports accelerated HTML delivery both within and across browser sessions but will not accelerate embedded object delivery across browser sessions. Any ACE configuration that uses delta optimization requires that the client browser support both cookies and JavaScript-based DHTML.

FlashForward and Repeat Visits

Most users will first experience FlashForward acceleration benefits on their first repeat visit to a given page. This "First Repeat Visit" represents the actual first visit for clients after the ACE cache is primed.

The examples in this section assume that the ACE was just installed and include an extra initial step required to prime the ACE cache with FlashForward objects (with or without delta optimization). The examples in this section show that, for the very first visit to the URL, the client will gain the FlashForward benefits on the second and subsequent repeat visits.

This three-visit requirement only applies to the client that actually primes the ACE cache with *new* (not modified) objects from the origin server (that is, the very first client that requested the new object). All other clients will automatically take advantage of the FlashForward benefits on the first repeat visit and later. This benefit occurs because other clients will fetch FlashForwarded objects on the very first visit; that is, the cookie-drop page (if delta optimization is configured) or the plain HTML page (if delta optimization is not configured). These clients will contain the transformed URLs that point to FlashForwarded objects in the ACE cache.

How FlashForward Works With Delta Optimization

This section describes how FlashForward works with delta optimization over a series of three visits to a page by a client browser.

Visit 1: Priming the Cache

This section describes the process for priming the cache on an ACE that has just been installed (its cache is empty). No base files or objects are yet in the ACE cache. The very first request for a given URL overall (not the first visit per client) is used to prime the ACE cache as follows:

1. The client requests a URL. The ACE proxies it to the origin server.
2. The origin server delivers the HTML page to the ACE.
3. The ACE retrieves the page, parses through the HTML page looking for references to embedded objects, and checks if the referenced objects are currently cached locally. In this case, none of the embedded objects referenced in the HTML are cached in the ACE because for this first visit, the ACE cache has not been primed with the embedded objects. The ACE prepends the ACE cookie through JavaScript and delivers the page compressed but otherwise unaltered (a standard cookie-drop page is created and delivered without the FlashForward function taking place).
4. The ACE creates the base page as usual because this visit is the first to the URL by a user.
5. The client retrieves the compressed "cookie-drop" page, parses the HTML looking for references to embedded objects, and requests the embedded objects directly from the origin server through HTTP GET requests.
6. Because the ACE is a proxy, the HTTP GETs are passed through the ACE to the origin server and the subsequent object responses (HTTP "200 OK") from the origin server are passed through the ACE to the client. The ACE caches all cacheable objects as they pass through, which primes the cache. The ACE uses the client's HTTP GET requests to populate the cache rather than resorting to some complex cache pre-population capability. This process occurs only the first time that new objects are delivered from the server to the client (only when the ACE determines that the original HTML references objects that are not in the cache at the time of the HTML request). The ACE cache is now primed.
7. The client browser retrieves and caches the original objects referenced in the HTML, as delivered by the origin server.

First Repeat Visit: The FlashForward Transformation

This section describes the process for the first repeat visit:

1. In a new browser session, the client requests the same (or similar) URL.
2. The ACE sees its cookie and knows that the client can support optimized responses.
3. The ACE proxies the request to the origin server.
4. The origin server creates and delivers the HTML page to the ACE.
5. The ACE parses through the HTML page looking for references to embedded objects and checks if the referenced objects are cached locally. If so, the ACE checks to see if the cached objects can be FlashForwarded (it determines if it is configured to FlashForward the object type). If the objects are both cacheable and can be FlashForwarded, the ACE issues HTTP IMS requests to the origin server to determine whether or not the cached objects are still valid.

Note The ACE does not issue If-Modified-Since (IMS) requests on every client request; it does not check object freshness with the origin server every time it gets a request from a client. Instead, the ACE uses the cache freshness settings in the optimization parameter map to determine how often it should issue IMS requests. For example, if you configure the cache expiration setting to be 10 minutes, the ACE will issue IMS requests for that object type only every 10 minutes.

   a. If the origin server responds with an HTTP 304 “Not Modified” status code, the ACE renames the already cached object by adding a version to the object name (URL) and adds a long-lived “Expires” HTTP response header to it (the default expiration date is greater than 20 years). This object has just undergone the FlashForward transformation. The ACE uses the 304 response information to eliminate the need for the client to issue IMS requests to validate the freshness of this object. This process eliminates the WAN round-trip time associated with IMS/304 traffic and results in an accelerated page download that is seen by the client.
   b. If the origin server responds with an HTTP 200 “OK” status code (with the modified object), the ACE caches the modified object, renames the newly cached object by adding a version to the object name (URL), and

adds a long-lived “Expires” HTTP response header to it (the default expiration date is greater than 20 years). This object has just undergone the FlashForward transformation.

6. The ACE next rewrites the HTML delivered by the origin server so that the embedded object references (URLs) point to the new FlashForward-transformed names (the version-added names that represent the objects in the ACE cache) rather than the original objects on the origin server.
7. The ACE compares the rewritten HTML to the base page to create a delta page (it performs delta optimization on the rewritten HTML). Delta optimization is used to deliver the changes in the HTML content and the changes in the URL references that result from the FlashForward process. This is the key to how FlashForward and delta optimization work together.
8. The ACE compresses and delivers the delta page to the client.
9. The client retrieves the delta page and reconstructs the rewritten HTML from the delta page and the base page.
10. The client browser parses through the HTML looking for references to embedded objects. These references now point to the FlashForwarded objects cached in the ACE.
11. The client requests the FlashForwarded objects from the ACE; the ACE delivers the FlashForwarded objects to the client.
12. The client browser retrieves and caches the FlashForwarded objects.

Second Repeat Visit: Gaining the Benefit

This section describes the process for the second repeat visit:

1. In a new browser session, the client requests the same (or similar) URL.
2. The ACE sees its cookie and knows that the client can support condensed responses.
3. The ACE proxies the request to the origin server.
4. The origin server creates and delivers the HTML page to the ACE.
5. The ACE parses through the HTML looking for references to embedded objects and checks if the referenced objects are cached locally. If so, the ACE checks if the cached objects can be FlashForwarded (determines if the ACE

is configured to FlashForward the object type). If the objects are both cacheable and can be FlashForwarded, the ACE issues HTTP IMS requests to the origin server to determine whether or not the cached objects are still valid.

a. If the origin server responds with an HTTP 304 “Not Modified” status code, the ACE knows that the already renamed FlashForwarded object currently in the cache is still valid.

b. If the origin server responds with an HTTP 200 “OK” status code (with the modified object), the ACE caches the modified object, renames the newly cached object by adding a version to the object name (URL), and adds a long-lived “Expires” HTTP response header to it (the default expiration date is greater than 20 years). This object has just undergone the FlashForward transformation.

6. The ACE next rewrites the HTML delivered by the origin server so that the embedded object URLs point to the FlashForward-transformed names. If the object did not change, the object name would be the same object name previously referenced in visit 2 (the same name under which the client has cached it in the browser). If the object was modified, the name would be the new version-attached name (it represents an object not currently in the client's browser cache).

7. The ACE compares the rewritten HTML to the base page to create a delta page (it performs delta optimization on the rewritten HTML). Delta optimization is used to deliver both changes in the HTML content and changes in URL references resulting from the FlashForward process.

8. The ACE compresses and delivers the delta page to the client.

9. The client retrieves the delta page and reconstructs the rewritten HTML from the delta page and the base page.

10. The client’s browser parses through the HTML looking for references to embedded objects. These references now point to the FlashForwarded objects cached in the ACE.

11. Because of the Expires header added to the embedded objects within the browser cache, the browser now issues HTTP GET requests only for objects referenced in the HTML that are not already cached in the browser (only the changed objects will be requested through HTTP GETs). No HTTP GET requests of any kind will be issued for any of the FlashForwarded objects already cached because they are known to still be fresh when they have the long-lived Expires date on them. FlashForward enables the ACE to determine embedded object freshness dynamically and explicitly communicate this

information to the client so that the client does not waste valuable time and bandwidth issuing requests to validate object freshness. This process accelerates the page download because it eliminates all IMS requests for objects known by the ACE to still be valid.

The next section discusses how FlashForward works without delta optimization enabled.

How FlashForward Works Without Delta Optimization

This section describes how FlashForward works without delta optimization over a series of two visits to a page by a client browser.

Visit 1: Priming the Application Appliance Cache

This section describes the process for an ACE that has just been installed (its cache is empty). The very first request for a given URL overall (not the first visit per client) is used to prime the ACE cache as follows:

1. The client requests a URL. The ACE proxies it to the origin server.
2. The origin server delivers the HTML page to the ACE.
3. The ACE retrieves the page, parses through the HTML page looking for references to embedded objects, and checks if the referenced objects are currently cached locally. In this case, no embedded objects that are referenced in the HTML are cached in the ACE because the ACE cache has not been primed with the embedded objects. The ACE delivers the page compressed but otherwise unaltered as usual (a standard cookie-drop page is created and delivered without the FlashForward function taking place). Because delta optimization is not configured, no cookie-drop JavaScript will be added to the page.
4. The client retrieves the compressed HTML page, parses the HTML page looking for references to embedded objects, and requests the embedded objects directly from the origin server through HTTP GET requests.
5. Because the ACE is a proxy, the HTTP GETs are passed through the ACE to the origin server and the subsequent object responses (HTTP “200 OK”) from the origin server are passed through the ACE to the client. The ACE caches all cacheable objects as they pass through. The ACE uses the client’s HTTP GET requests to populate the cache rather than having to resort to some

complex cache pre-population capability. This process occurs only the first time that new objects are delivered from the server to the client (when the original HTML references objects that are not in the cache at the time of the HTML request). The ACE cache is now primed.

6. The client browser retrieves and caches the original objects referenced in the HTML as delivered by the origin server.

First Repeat Visit: The FlashForward Transformation

This section describes the process for the first repeat visit:

1. In a new browser session, the client requests the same (or similar) URL.
2. The ACE proxies the request to the origin server.
3. The origin server creates and delivers the HTML page to the ACE.
4. The ACE parses through the HTML page looking for references to embedded objects and checks if the referenced objects are cached locally. If so, the ACE checks if the cached objects can be FlashForwarded (it determines if it is configured to FlashForward the object type). If the objects are both cacheable and can be FlashForwarded, the ACE issues HTTP IMS requests to the origin server to determine if the cached objects are still valid.
   a. If the origin server responds with an HTTP 304 “Not Modified” status code, the ACE renames the already cached object by adding a version to the object name (URL), and adds a long-lived “Expires” HTTP response header to it (the default expiration date is greater than 20 years). This object has just undergone the FlashForward transformation. The ACE uses the 304 response information to eliminate the need for the client to issue IMS requests to validate the freshness of this object. This process eliminates the WAN round-trip time associated with IMS/304 traffic, which results in an accelerated page download that is seen by the client.
   b. If the origin server responds with an HTTP 200 “OK” status code (with the modified object), the ACE caches the modified object and renames the newly cached object by adding a version to the object name (URL), and adds a long-lived “Expires” HTTP response header to it (the default expiration date is greater than 20 years). This object has just undergone the FlashForward transformation.

5. The ACE next rewrites the HTML delivered by the origin server so that the embedded object references (URLs) point to the new FlashForward-transformed names rather than the original objects on the origin server.
6. The ACE compresses and delivers the rewritten HTML page to the client.
7. The client browser retrieves the rewritten HTML and parses through it looking for references to embedded objects. These references now point to the FlashForwarded objects cached in the ACE.
8. The client requests the FlashForwarded objects from the ACE; the ACE delivers the FlashForwarded objects to the client.
9. The client browser retrieves and caches the FlashForwarded objects.

Second Repeat Visit: Gaining the Benefit

This section describes the process for the second repeat visit:

1. In a new browser session, the client requests the same (or similar) URL.
2. The ACE proxies the request to the origin server.
3. The origin server creates and delivers the HTML page to the ACE.
4. The ACE parses through the HTML page looking for references to embedded objects and checks if the referenced objects are cached locally. If so, the ACE checks if the cached objects can be FlashForwarded (it determines if it is configured to FlashForward the object type). If the objects are both cacheable and can be FlashForwarded, the ACE issues HTTP IMS requests to the origin server to determine if the cached objects are still valid.
    a. If the origin server responds with an HTTP 304 “Not Modified” status code, the ACE knows that the already renamed FlashForwarded object currently in the cache is still valid.
    b. If the origin server responds with an HTTP 200 “OK” status code (with the modified object), the ACE caches the modified object and renames the newly cached object by adding a version to the object name (URL), and adds a long-lived “Expires” HTTP response header to it (the default expiration date is greater than 20 years). This object has just undergone the FlashForward transformation.

5. The ACE next rewrites the HTML delivered by the origin server so that the embedded object URLs point to the FlashForward-transformed names. If the object did not change, the object name would be the same object name previously referenced in visit 2 (the same name under which the client has cached it in the browser); but if the object was modified, the name would be the new version-attached name (it represents an object not currently in the client's browser cache).
6. The ACE compresses and delivers the rewritten HTML to the client.
7. The client browser retrieves the rewritten HTML and parses through it looking for references to embedded objects. These references now point to the FlashForwarded objects cached in the ACE.
8. Because of the Expires header added to the embedded objects within the browser cache, the browser now issues HTTP GET requests only for objects referenced in the HTML that are not already cached in the browser (only the changed objects will be requested through HTTP GETs). No HTTP GET requests of any kind will be issued for any of the FlashForwarded objects already cached because they are known to still be fresh when they have the long-lived Expires date on them. FlashForward enables the ACE to determine embedded object freshness dynamically and explicitly communicate this information to the client so that the client does not waste valuable time and bandwidth issuing requests to validate object freshness. This process accelerates the page download because it eliminates all IMS requests for objects known by the ACE to still be valid.

Embedded Objects Referenced Within JavaScript and Not Within HTML

To find references to embedded objects, the ACE parses for img, script, href, and background tags within the HTML. The ACE will not find references to embedded objects within JavaScript.

Because the ACE caches all cacheable embedded objects during the priming stage, it will still FlashForward cacheable JavaScript-embedded objects by adding an Expires header to these objects without renaming them. The date and time associated with the added Expires header are defined by the ACE cache freshness settings in the **cache ttl** command in parameter map optimization mode.

To control the period of time that objects in the client's browser remain fresh, use the **expires-setting** command in parameter map optimization mode.

For details, see Chapter 3, Configuring an Optimization HTTP Parameter Map section.

How FlashForward Works With CDN URLs

The ACE can apply FlashForward object acceleration to objects and URLs transformed by content delivery networks (CDNs). Embedding the original name in the CDN URL is required to allow the CDN edge cache to identify and fetch the object from the origin server the first time that it is requested.

As an example, consider the following typical URL that has been transformed by Akamai:

```
http://a484.g.akamai.net/f/484/868/1h/www.hilton.com/en/hi/media/images/tabs/tab_0.gif
```

Also consider the following typical URL that has been transformed by Speedera:

```
http://gateway.speedera.net/www.gateway.com/images/cp/banners/homepage_bnr_broadband01.gif
```

In both situations, the CDN-modified URLs consist of a CDN ID portion, followed by the original URL. The CDN ID portion appears in bold in the following examples:

```
http://a484.g.akamai.net/f/484/868/1h/www.hilton.com/en/hi/media/images/tabs/tab_0.gif
```

```
http://gateway.speedera.net/www.gateway.com/images/cp/banners/homepage_bnr_broadband01.gif
```

Because CDN-modified URLs embed the origin server URLs within them, the ACE is able to extract the original portion of the URLs needed for FlashForward object validation.

You can identify whether Akamai or Speedera are being used on the content being FlashForwarded. The feature enables the ACE to identify and parse through such CDN URLs to extract the original URL portion and to perform embedded object validation requests with the origin server as required by FlashForward. FlashForward transformation occurs as usual; for example, the ACE appends a unique ID to the CDN URL. The FlashForward-transformed CDN URL changes whenever the object is modified. The ID is based on an MD5 hash of the object, so if the object changes, the hash changes, and the URL changes.

FlashForward-transformed CDN URLs look like the following for Akamai:

```
http://a484.g.akamai.net/f/484/868/1h/www.hilton.com/en/hi/media/images/tabs/
tab_0_vtmmi14xg2fvmlkxsxuk0ty1xd_acc_V01.gif
```

In this example, tab_0.gif has been replaced with the FlashForward-transformed object name:

```
tab_0_vtmmi14xg2fvmlkxsxuk0ty1xd_ACC_V01.gif
```

Similarly, for Speedera, the FlashForward-transformed URL looks like the following:

```
http://gateway.speedera.net/www.gateway.com/images
/cp/banners/homepage_bnr_broadband01_5f1fdnjgsaigblyav4f42wnb1g_ACC_V0
1.gif
```

In this example, homepage_bnr_broadband01.gif has been replaced with the FlashForward-transformed object name:

```
homepage_bnr_broadband01_5f1fdnjgsaigblyav4f42wnb1g_ACC_V01.gif
```

The ACE can FlashForward objects transformed by Akamai and Speedera and enable these CDNs to deliver and cache the FlashForward-transformed objects. The end result is that clients are able to fetch FlashForwarded objects from the CDN, and clients no longer need to issue IMS requests to the CDN on subsequent session requests.

To modify the cache key, use the **cache key-modifier** command in parameter map optimization mode, which is identical to the canonical URL definition in use. The **cache key-modifier** command is based on regular expressions. For details, see Chapter 3, Configuring an Optimization HTTP Parameter Map.

A configuration example for Akami is as follows:

```
host1/Admin(config)# class-map type http loadbalance match-any
AkamaiModifier_Classmap
host1/Admin(config-cmap-http-lb)# match http url
.*akamai\.net.*www(.*\.jpg)
host1/Admin(config-cmap-http-lb)# match http url
.*akamai\.net.*www(.*\.gif)
host1/Admin(config-cmap-http-lb)# match http url
.*akamai\.net.*www(.*\.jpeg)
host1/Admin(config-cmap-http-lb)# exit
host1/Admin(config)# action-list type optimization http ACT_LIST1
host1/Admin(config-actlist-optm)# no delta
host1/Admin(config-actlist-optm)# flashforward-object
```

```
host1/Admin(config-actlist-optm)# exit
host1/Admin(config)# parameter-map type optimization http
OPTIMIZE_PARAM_MAP1
host1/Admin(config-parammap-optmz)# cache key-modifier http://www$(1)
host1/Admin(config-parammap-optmz)# cache-policy request override-all
host1/Admin(config-parammap-optmz)# exit
host/Admin(config)# policy-map type optimization http first-match
L7OPTIMIZATION_POLICY
host/Admin(config-pmap-optmz)# class AkamaiModifier_Classmap
host1/Admin(config-pmap-optmz-c)# action ACT_LIST1 parameter
OPTIMIZE_PARAM_MAP1
```

The AkamaiModifier_Classmap class map matches images that have URLs transformed by Akamai (see the Layer 7 HTTP loadbalance class map definition). The parentheses in this example define a subexpression, that when matched, is used in the **cache key-modifier** command. Each "( )" expression is numbered (index starting at 1) and can be used in any way by using the expression $(number)$. For more details on using subexpressions, see Chapter 3, Configuring an Optimization HTTP Parameter Map for details. Table 3-3 lists the parameter expander functions that you can use.

In this example, the cache key is modified to strip away the Akamai portion of the URL to ensure that f1==f2, and FlashForward operates correctly. However, CDN may not fetch the object from the ACE for a considerable period of time, in which case FlashForward is not effective.

A similar configuration example for Speedera is as follows:

```
host1/Admin(config)# class-map type http loadbalance match-any
SpeederaModifier_Classmap
host1/Admin(config-cmap-http-lb)# match http url
.*speedera\.net.*www(.*\.gif)
host1/Admin(config-cmap-http-lb)# exit
host1/Admin(config)# action-list type optimization http ACT_LIST2
host1/Admin(config-actlist-optm)# no delta
host1/Admin(config-actlist-optm)# flashforward-object
host1/Admin(config-actlist-optm)# exit
host1/Admin(config)# parameter-map type optimization http
OPTIMIZE_PARAM_MAP2
host1/Admin(config-parammap-optmz)# cache key-modifier http://www$(1)
host1/Admin(config-parammap-optmz)# cache-policy request override-all
host1/Admin(config-parammap-optmz)# exit
host/Admin(config)# policy-map type optimization http first-match
L7OPTIMIZATION_POLICY
host/Admin(config-pmap-optmz)# class SpeederaModifier_Classmap
host1/Admin(config-pmap-optmz-c)# action ACT_LIST2 parameter
OPTIMIZE_PARAM_MAP2
```

This configuration is similar to the Akamai example configuration, except that the URLs that match this class are different.

Where to Go Next

Proceed to Chapter 2, Configuring an Optimization HTTP Action List, to configure an optimization HTTP action list. An action list groups a series of individual application acceleration and optimization functions that apply to a specific type of operation.