



AVS Description

The Cisco Application Velocity System (AVS) application appliance accelerates enterprise applications, resulting in increased employee productivity, enhanced customer retention, and increased online revenues. The application appliance enables enterprises to optimize network performance and improve access to critical business information. The application appliance accelerates the performance of web applications, including customer relationship management (CRM), portals, and online collaboration by up to 10 times.

The application appliance's unique application acceleration benefits are enabled by the following Condenser Application Accelerator technologies, which are described in these sections:

- [Delta Optimization, page 2-2](#)
- [Adaptive Dynamic Caching, page 2-5](#)
- [FlashForward Object Acceleration, page 2-6](#)
- [Just-In-Time Object Acceleration, page 2-7](#)
- [Smart Image Optimization, page 2-8](#)
- [Smart Redirect, page 2-11](#)
- [SSL Termination and Proxy, page 2-11](#)
- [Fast NTLM Authentication, page 2-12](#)
- [Server Connection Offload, page 2-12](#)
- [Text Compression, page 2-12](#)
- [FlashConnect, page 2-12](#)

Additional features are described in the “[Other Features](#)” section on page 2-13.

AppScope Performance Monitoring is described in the “[AppScope Performance Monitoring](#)” section on page 2-18.

The AppScreen Web Application Firewall is described in the “[AppScreen Web Application Firewall](#)” section on page 2-18.

Finally, this chapter includes a description of how the Condenser works in the “[Operation](#)” section on page 2-19.

Delta Optimization

This section includes the following topics:

- [Delta Optimization Overview, page 2-2](#)
- [Web Browser Support, page 2-3](#)
- [Web Server Support, page 2-4](#)
- [Web Content Support, page 2-4](#)
- [Configurable Condensation Modes, page 2-5](#)

Delta Optimization Overview

The application appliance enables enterprises to dynamically update client browser caches directly with content differences, or deltas, resulting in faster page downloads, improved employee productivity, and increased online revenues.

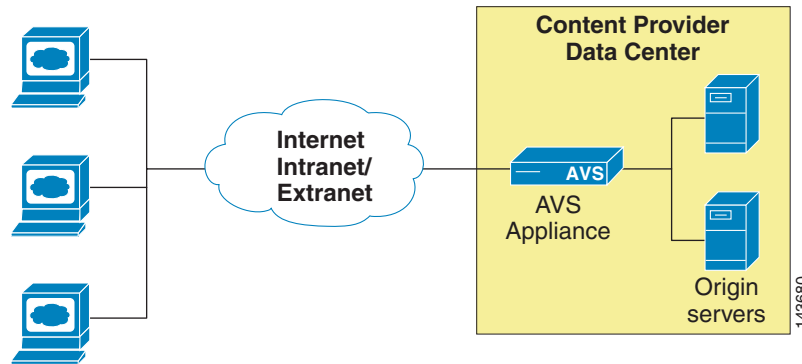
Many web pages are created dynamically, such that each request produces different content. The differences in content could result from different ad payloads being rotated in and out of the page, or from more deeply rooted HTML content changes (for example, changing stock quotes, changing news headlines, and so on). Because these dynamically created pages change with every 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 the document is requested, 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 ad payloads for each request. In addition, news headlines on this page may change during the day. Without condensation technology, a user must download the entire 70 KB with every subsequent visit to the page, even though the changes in the underlying HTML comprise only about 2 KB of the page.

Condensation technology allows the application appliance 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. As a result, 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, resulting in a factor of 35 reduction in outbound bandwidth requirements and delivering a factor of up to 10 content acceleration for the client. With condensation, only the deltas between subsequently requested pages are sent to users. These deltas, which are encoded by using dynamic HTML, enable the application appliance to directly update the client's browser cache, much like an origin server updates a traditional edge cache.

[Figure 2-1](#) shows how the application appliance fits into the network topology.

Figure 2-1 Simplified AVS Topology



The application appliance is located in the path of the content delivered from a content origination server (web server) to the browser, but close to the server. It is generally deployed as a transparent proxy through a load balancer. The application appliance observes and modifies the content flowing through it to achieve bandwidth savings and increase user download performance.

Web Browser Support

The application appliance supports traffic directed to and from all web browsers. This includes support for HTTP 1.0 and HTTP 1.1 protocols. HTTP 1.1 support includes chunking using the “Transfer-Encoding: Chunked” HTTP request header as described in section 14.40 of [RFC 2068](#), “Hypertext Transfer Protocol -- HTTP/1.1.”

Full optimization is supported for all major browsers on Windows 98, Windows NT 4, Windows 2000, Windows ME, and Windows XP if cookies and JavaScript are enabled in these versions:

- Microsoft Internet Explorer versions 4.0 and later
- Netscape Communicator versions 4.05 and later
- AOL browser versions 5.0 and later

All other browsers and configurations are supported without Delta Optimization.

When the user requests a page, the application appliance identifies the user’s browser type and applies those optimizations supported by the browser. For example, if the application appliance determines that a particular browser does not support gzip compression, it automatically applies all other optimizations, including Delta Optimization and FlashForward, without applying gzip compression. For details on controlling condensation support for specific browsers, see the “[useragent.conf](#)” section on page 5-39.

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

Cookie and JavaScript support are detected as follows. On the user’s first visit to a site, the application appliance inserts JavaScript probe code in the page delivered to the user. When executed by the browser, this JavaScript code creates an application appliance cookie on the user’s system. If the browser does not support JavaScript or it is disabled, or if cookies have been disabled, then the cookie creation fails; otherwise, the application appliance cookie is successfully created. When the user visits the site the second time, the presence of the cookie indicates that the browser supports JavaScript and cookies and signals to the application appliance to deliver condensed content to the user.

**Note**

The application appliance also handles the case where a user subsequently reconfigures their browser to disable JavaScript. For more information, see the [“Cookie Usage” section on page 2-17](#).

The application appliance can also detect browsers that can handle gzip-encoded content (it gzip compresses the response content, even if the content is not condensed).

Web Server Support

The application appliance supports all web servers and web application servers that support the HTTP 1.0 or 1.1 protocols on all platforms. HTTP 1.1 support includes support for 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 application appliance supports all web content without modification to the content or the server software. Not all web content is suitable for condensation, however. Uncondensable content is passed through unmodified. The following sections describe the limitations to condensation support:

- [Character Set Support, page 2-4](#)
- [Unsupported HTML Elements, page 2-4](#)

Character Set Support

Condensation occurs only on HTML content that uses the iso-8859-1 character set because the application appliance supports only single-byte characters. All other types of content are passed through uncondensed.

The character set type is detected by examining the Content-Type HTTP response header, which looks like this:

```
Content-Type: text/html; charset=iso-8859-1
```

If no character set specification is present, it is assumed to be iso-8859-1.

Unsupported HTML Elements

Certain types of HTML content are not condensable. Inline frames (IFRAME tags), external script includes (SCRIPT tags including the SRC attribute), and embedded object (OBJECT tags) cannot be condensed. While pages containing these elements are condensed, these elements within the page are passed through uncondensed.

The application appliance uses its content inspection feature to examine pages for other elements that make pages uncondensable and automatically excludes those pages from condensation if it finds any of those elements. The application appliance performs content inspection by examining the first 1 KB of each page by using a content sensing algorithm from the W3C library. Content inspection is enabled by default and may be disabled if desired.

Content inspection can be a useful feature, but it can slow the overall performance of the application appliance, so we recommend that you do not use this feature unless absolutely necessary. Instead, we recommend that you remove unsupported elements from your web pages, or exclude such pages from condensation.

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 avoids problems related to JavaScript dependencies and execution order.

Configurable Condensation Modes

The condensation mode specifies whether the web pages to be condensed are common to all users or personalized for individual users, which determines what kind of page deltas are generated by the application appliance.

The application appliance supports two condensation modes:

- All-user condensation
- Per-user condensation

In the all-user condensation mode, the delta is generated against a single base file that is shared by all users of the URL. The all-user condensation 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 condensation 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 condensation 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 condensation. However, a copy of the base page that is delivered to each user has to be kept in the application appliance cache and this increases the requirements on disk space for the application appliance cache. The per-user condensation mode is useful for content privacy because base pages are not shared among users.

Adaptive Dynamic Caching

Adaptive dynamic caching enables the application appliance to fulfill requests for dynamic or personalized information, thus offloading application servers and databases. Adaptive dynamic caching not only significantly improves application response time but also reduces 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 application appliance to monitor server load in real-time and make intelligent “closed-loop” content expiration decisions. This maximizes site performance and uses hardware resources most efficiently even during peak traffic load.

Adaptive dynamic caching enables the application appliance to cache dynamic content such as content created by an active server pages (ASP) script or application server. 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 application appliance allows you to cache this kind of content for some period of time.

Adaptive dynamic caching has these features:

- **Cache parameterization.** Can differentiate a response by more than the URL and its query parameters. It allows the use of other parameters such as cookie values, HTTP header values, and the HTTP method used. This feature allows the application appliance to cache multiple responses for a single URL, depending on the specified cache parameters. This feature applies to both static (FlashForward) and dynamic caching.
- **Cacheability rules.** Allows IF/ELSE conditional rules to determine if a dynamic response is cacheable. The rules can examine URL query parameters, cookies, HTTP headers, and the HTTP method used. This feature applies to both static (FlashForward) and dynamic caching.
- **Expanded expiration rules.** Can set the automatic expiration of cached content based on time or through performance assurance with load-based expiration.
- **Delta Cache.** When the original HTML content is in the dynamic cache and a rebase has not occurred, the delta content can be stored in cache, which is a memory-only object.

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 web site or application. It is configurable at the Application Class level, which allows different caching rules to be applied to different sets of URLs. For guidelines on when to use dynamic caching and the steps required to implement it, refer to the [“Dynamic Caching Configuration Guide” section on page 5-34](#).

For reference information on configuring dynamic caching in the fgn.conf file, see the [“Adaptive Caching Configuration” section on page 5-18](#).

FlashForward Object Acceleration

FlashForward object acceleration extends the application appliance’s bandwidth usage reduction and download acceleration benefits to objects that are embedded within HTML pages. This feature combines 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 application appliance, 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 only requests objects known to be new or modified at the time of the HTML request. This technique results in significantly accelerated page downloads as realized by the client and reduced upstream traffic associated with object validation requests.

FlashForward eliminates the network delays associated with embedded web objects such as images, style sheets, JavaScript files, and so on. Without the application appliance, 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 many objects because the objects cannot be rendered until the client-to-server round-trips are completed. In addition, this approach wastes significant upstream bandwidth.

FlashForward places the responsibility for validating object freshness on the application appliance, 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 application appliance has determined to be valid. Working together with Delta Optimization technology, 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 on how FlashForward operates, see the [“FlashForward Operation” section on page 2-23](#). To configure FlashForward, use the FlashForward and FlashForwardObject policy keywords, as described in [Table 5-3 on page 5-12](#).

Just-In-Time Object Acceleration

Just as FlashForward accelerates delivery of embedded cacheable objects, just-in-time object acceleration enables acceleration of noncacheable embedded objects, which results in improved application response time. This feature eliminates the need for users to download these objects on each request. Instead, the application appliance automatically tracks the freshness of each object in real time. If a requested object has not changed, the application appliance instructs the client to use its cached version of the object. If an object has changed, the application appliance delivers it to the client. The application appliance delivers the object only if it determines that the object has changed, guaranteeing the optimal application response times for all users.

Static content like images is handled with FlashForward, and dynamic HTML is handled 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:

- HTML content is dynamic and larger than the maximum condensable page size (250 KB).
- Content is marked by the origin server as expired or not cacheable, just in case it might occasionally change, but actually it does not change often.

We recommend that you use just-in-time object acceleration with compression, for best results.

This feature is implemented as follows. The browser first requests an object and the application appliance fetches it on behalf of the browser request. The application appliance 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 can be used to identify different versions of the same object. All subsequent requests from the browser include this ETag header, which the application appliance compares with a recomputed MD5 hash of the latest origin server content. If the content has not changed, then the application appliance returns a 304 (Not Modified) response and no data. If the content has changed, the application appliance retrieves the new content and resets the ETag.

To configure just-in-time object acceleration, use the DynamicETag policy keyword, as described in [Table 5-3 on page 5-12](#).

Smart Image Optimization

Smart image optimization reduces JPEG and PNG image file sizes while optimizing image quality, which results in faster image download times, faster page renders, and more efficient bandwidth utilization.

Existing image compression techniques uniformly compress images so that areas of rich detail are compressed as much as areas of low detail. These approaches reduce image sizes but severely degrade image quality.

Smart image optimization recompresses only low detail areas within images to ensure that image sizes are significantly reduced (up to 90%) while maintaining rich visual detail. No changes to client software, server content, or server configuration are required.

Smart image optimization works with FlashForward to enable the fastest image optimization and delivery available today.

Smart image optimization is applied intelligently, and is not used for small images such as thumbnails, or when optimization reduces the file size by less than 10%. Only images that can be cached are optimized. If content includes very large images that you want optimized, you may need to increase the MaxCacheableObjectSize (which is set to 250 KB by default) to ensure that they are cached.

For detailed configuration information, see the [“Image Optimization Configuration”](#) section on [page 5-24](#).

It is possible that you could see the difference in image quality between optimized and unoptimized images; however, unless you are looking for a problem, little difference is noticeable. The JPEG images in [Figure 2-2](#), [Figure 2-3](#), [Figure 2-4](#), and [Figure 2-5](#), show before and after versions, where the image size has been reduced by at least 50% in each case.

Figure 2-2 Unoptimized Image Example 1 (67 KB)



Figure 2-3 Optimized Image Example 1 (31 KB)



Figure 2-4 *Unoptimized Image Example 2 (43 KB)*



143763

Figure 2-5 *Optimized Image Example 2 (19 KB)*



143762

Smart Redirect

Some applications and content management systems enable enterprises to automatically redirect users from one page to another through HTML META tags. Using HTML META tag page redirections can cause poor download times because it forces the browser to issue freshness validation requests for every object in the redirected page, which could result in potentially significant page download delays.

Smart redirect enables the application appliance to automatically and transparently convert HTML META tag redirections into more efficient HTTP header-based redirections. Using this feature eliminates the need for unnecessary freshness validation requests and results in significantly faster page response times, without sacrificing the META tag redirection flexibility enabled by many enterprise applications.

**Note**

This feature is not suitable in situations where the intent of the META tag redirect is simply to reload the page, rather than redirect to a different page.

To configure smart redirect, use the MetaRefreshTo302 policy keyword, as described in [Table 5-3 on page 5-12](#).

SSL Termination and Proxy

The application appliance supports condensation of SSL (Secure Sockets Layer) content. This configurable option enables the application appliance to deliver condensation for selected SSLv3 encrypted content. With this feature, you can configure the product as an SSL terminator, SSL proxy, and/or a Condenser.

As an SSL terminator, the application appliance operates as an application-layer proxy that communicates with end-user clients through HTTPS (SSL) and with the origin server within the data center through clear-text HTTP. SSL termination enables client-to-appliance security through SSL encryption, leaving appliance-to-server traffic in the clear through HTTP.

As an SSL proxy, the application appliance operates as an application-layer proxy that communicates with end-user clients and with the origin server within the data center through HTTPS (SSL). All traffic between the client and the server is encrypted through SSL; no clear-text traffic is seen. SSL proxying enables end-to-end client-to-server security through SSL encryption.

In a single application appliance environment where the application appliance is deployed as a terminator, the application appliance terminates an SSL session, decrypts the request, passes the decrypted request to the origin server, retrieves, optimizes, and encrypts the server response with SSL, and delivers the encrypted response to the client. Where the application appliance is deployed as a proxy, it decrypts the request to inspect the query, reencrypts the request, transparently proxies it through SSL to the origin server, retrieves, decrypts, optimizes, and reencrypts the server response with SSL, and delivers the encrypted response to the client.

SSL termination and proxy are disabled by default and must be explicitly enabled. They are enabled through destination mapping and other configuration parameters in the httpd.conf and fgn.conf files. See the following sections for specific information:

- For information on enabling SSL features in the httpd.conf file, see the [“SSL Configuration Entries” section on page 5-37](#).
- For information on configuring destination mapping and Condensation policy in the fgn.conf file, see the [“SSL Configuration” section on page 5-33](#).

Fast NTLM Authentication

The application appliance significantly improves the overall application performance in NT LAN Manager (NTLM)-enabled environments by eliminating redundant NTLM authentication traffic associated with object validation requests. This is similar to how the application appliance improves SSL performance by eliminating unnecessary object validation requests that require costly SSL handshakes.

Server Connection Offload

Server connection offload enables the application appliance to optimize TCP-level efficiency by “pooling” a number of appliance-to-server TCP connections. Pooling enables multiple users to share a single TCP connection from the application appliance to the origin server, eliminating the need to create a new TCP connection for each transaction. Connection sharing increases the application appliance performance and scalability by reducing network-level overhead associated with TCP connection management.

Text Compression

Typically, standard text compression provides only modest improvements in real-world conditions and is not sufficiently powerful or robust for enterprise requirements. The application appliance uses industry-standard gzip compression to further reduce the byte size of delta optimized pages. Reducing page sizes is key to accelerating download times.

The application appliance compresses response content even if the content is not able to be condensed.

You can select either the gzip or deflate compression type by using the `CompressionMethod` keyword, listed in [Table 5-1 on page 5-2](#).

FlashConnect

Like FlashForward, FlashConnect allows the application appliance to reduce the bandwidth usage and accelerate downloading of objects that are embedded within HTML pages.

FlashConnect dynamically renames embedded objects by adding a prefix and changing the hostname, making the objects appear to reside on different hosts even though they may all reside on a single host. FlashConnect makes the browser open separate connections to the origin server for each object, which increases the network performance because the objects are retrieved in parallel, rather than one after another.

To use this feature, you must also configure DNS so that all requests for the rewritten object URLs are resolved back to the application appliance node (that rewrote them initially).

FlashConnect is disabled by default and must be explicitly enabled by specifying the policy keywords `FlashConnect` (for container pages) and `FlashConnectObject` (for embedded objects), as described in [Table 5-3 on page 5-12](#).

Additionally, you can limit the number of artificial hosts that FlashConnect uses by specifying a limit with the `FlashConnectLimit` global keyword. The default limit is four. Finally, you can use the `FlashConnectPrefix` global keyword to set the prefix that is added to embedded object URLs. The default prefix is `flashconnect`. These keywords are described in [Table 5-1 on page 5-2](#).

Other Features

The application appliance includes other features described in these sections:

- [Class-Based Condensation, page 2-13](#)
- [Base File Management, page 2-13](#)
- [Canonical URL, page 2-14](#)
- [Base File Selection Policy, page 2-14](#)
- [Anonymous Base Files, page 2-14](#)
- [Smart Rebasing, page 2-15](#)
- [MIME-Type Exclusion, page 2-16](#)
- [SNMP Support, page 2-16](#)
- [Destination Mapping, page 2-16](#)
- [Log File Management, page 2-17](#)
- [Cookie Usage, page 2-17](#)

Class-Based Condensation

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

**Note**

The application appliance requires GNU POSIX regular expression syntax. For more information, see [Appendix F, “Regular Expressions.”](#)

Base File Management

The application appliance 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.

Canonical URL

The application appliance uses the canonical URL feature to modify a parameterized request to eliminate the “?” 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 application appliance uses this feature to map multiple parameterized URLs to a single canonical URL.

For example, the two URLs `http://www.servers.com/books?id=235` and `http://www.servers.com/books?id=576` would both be reduced to the URL `http://www.servers.com/books`. As a result, both of these 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 content or layout).

The canonical URL feature is enabled by default but can be overridden through the application appliance’s base file selection policy feature, which is described in the next section.

Base File Selection Policy

Base file selection policy is similar to the canonical URL feature, but it provides more flexibility in specifying a base file to be shared among a group of URLs.

The base file selection policy allows you to define, on a class-based basis, 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`, etc.

These URLs are parameterized by ID numbers within the path, rather than the typical ? 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.

You specify the base file selection policy by using the `CanonicalUrl` keyword in an Application Class in the `fgn.conf` file. For details, see the [“Base File Selection Policy” section on page 5-15](#).

Anonymous Base Files

The application appliance incorporates an anonymous base file feature to address user privacy concerns. This feature, which is an all-user condensation option, enables customers to use the application appliance nodes to deliver personalized confidential content such as online trading accounts, banking statements, business accounts, and so on. Customers typically 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 and/or non user 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 application appliance 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. Using 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 no longer represents any initial (potentially confidential) content as viewed by the first visitor to a particular URL.

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

You can configure the anonymity level (**m**). (Where **m**=0, no anonymity will be enabled). The application appliance 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***m**. Through extensive testing, we recommend an anonymity level of 2.

This feature is disabled by default and must be explicitly enabled by using the all-user condensation mode and specifying an additional BaseFileAnonLevel configuration keyword (corresponding to **m**, above) in the Application Class. For details, see the [“Application Class Specification” section on page 5-7](#).

For details about the statistical model used to calculate anonymity probabilities, refer to [Appendix E, “Anonymous Base File Statistical Model.”](#)

Smart Rebasing

Rebasing refers to the process of updating 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 condensation process, the base files are automatically updated as required.

Smart rebasing enables the application appliance 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.

An Application Class parameter called RebaseFlashForwardPercent provides a threshold control for rebasing based on the percent of FlashForwarded URLs in the response. Where the existing parameter, RebaseDeltaPercent, triggers rebasing when the delta response size exceeds the threshold as a percentage of base file size; this new parameter triggers rebasing when the difference between the percentages of FlashForwarded URLs in the delta response and the base file exceed the threshold. The default value for RebaseFlashForwardPercent is 50%.

With Smart Rebasing, the application appliance tracks the number of delta responses sampled, how many of the responses have a delta size bigger than the RebaseDeltaPercent threshold, and how many of the responses have too many FlashForwarded URLs (through the RebaseFlashForwardPercent threshold). When a reasonable sample size (10, to be specific) is reached, the application appliance looks at the percentages of delta responses that have exceeded the RebaseDeltaPercent and RebaseFlashForwardPercent thresholds. By default, rebasing is automatically triggered when either percentage exceeds 50%. Smart rebasing enables the application appliance 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 overall application appliance performance and content acceleration because it ensures that delta optimization occurs at all times, even when a rebase occurs.

MIME-Type Exclusion

Because some content providers choose to label text/html content as non-text/html MIME types (and vice versa) in the HTTP entity header field to prevent web crawlers, this feature allows you to explicitly configure specific MIME types as uncondensable. For example, suppose that you configure the web server to report JavaScript as MIME type “application/x-text.” With this feature configured to identify this MIME type as uncondensable, the application appliance will not condense responses for this MIME type. When you disable this feature, the application appliance will condense responses for this MIME type.

You can also list specific MIME types that are not to be compressed.

This feature is configured by listing MIME types that are not to be condensed or compressed in the `mimetypes.conf` file. If this file is empty, or does not exist, then all MIME types are considered condensable and compressible. For details on configuring MIME-type exclusion, see the [“mimetypes.conf” section on page 5-38](#).

No configuration is necessary for this feature. It is enabled automatically.

SNMP Support

The application appliance supports Simple Network Management Protocol (SNMP) for remote network management. The application appliance Management Information Base (MIB) is described in [Appendix B, “SNMP MIB.”](#)

This MIB is defined in the file `$AVS_HOME/perfnod/conf/fgn_cds_mib.mib`. A network management application can import this file. In this MIB, several traps are also defined that monitor the status of the application appliance. On the management application side, it is up to the administrator to define the severity of the traps and the corresponding actions.

Destination Mapping

The application appliance uses destination mapping to proxy requests to a specified destination IP address and port number or hostname and port. The application appliance supports wide scope mapping and host header mapping. Destination mapping is useful in clustered application appliance environments and these typical scenarios:

- the application appliance handles both an intranet and an Internet application
- the application appliance listens on one port and handles multiple internal sites
- the application appliance listens on multiple ports and redirects to multiple sites
- the application appliance resides behind a load balancer but redirects to a site through a proxy
- an SSL-terminating application appliance forwards decrypted requests to another application appliance
- an SSL-proxy application appliance forwards reencrypted requests to a site

In an SSL-terminating scenario, a load balancer redirects an SSL request to an application appliance SSL terminator through a virtual IP (VIP) configured on the load balancer. This VIP is bound to the real IP addresses of the application appliance.

The SSL terminator next proxies the request to an application appliance node through the load balancer. To achieve load balancing and failover of these requests, the SSL terminator must proxy the request to a new VIP or the same VIP on a different port bound to the application appliance IP address and port.

If a mapped destination port is not explicitly identified, the currently bound port will be used.

This feature is disabled by default and must be explicitly configured to enable it. For details, see the [“Destination Mapping Configuration” section on page 5-30](#).

Log File Management

The application appliance supports automatic log rotation and uploading logs to the Management Console database.

For details on the configuration directives in the fgn.conf file that control logging, see the [“fgn.conf” section on page 5-1](#).

For details on log file management and the format of data contained in the log files, refer to [Appendix A, “Logs.”](#)

Cookie Usage

To determine browser support for cookies and JavaScript, the application appliance 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 application appliance cookie will be created.

Upon a second request from the client, the application appliance looks for the application appliance cookie to verify client JavaScript and cookie support. If the application appliance sees the application appliance cookie, it assumes that the client supports JavaScript. If it does not see the application appliance 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 will not provide condensation for this user’s requests.

The application appliance cookie has the attributes listed in [Table 2-1](#).

Table 2-1 Application Velocity System Cookie Properties

Attribute	Value
Name	FGNCDN
Life	30 days
Domain	Same as the target site
Path	“/” (the entire site)
Value	A 128-bit randomly generated user ID. This ID uniquely identifies the user to the application appliance.

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

If JavaScript is disabled, the client executes HTML code within the <NOSCRIPT> tags to fetch a URL like “http://www.example.com/?cisco=removeCookie.” When the application appliance 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 application appliance cookie.

AppScope Performance Monitoring

AppScope measures true end-to-end application performance as seen by end users. AppScope also accurately determines both the server delay and network delay components associated with the user experience at the transaction level. AppScope’s unique Statistical Traffic Sampling technology enables an enterprise to statistically sample user requests, making AppScope highly scalable for high-traffic applications.

The AVS 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 database. This database allows an organization to use its own reporting tools, such as Crystal Reports, or to create its own custom performance monitoring reports.

For details on using AppScope Performance Monitoring and generating reports, see the “[AppScope Reports](#)” section on page 8-10.



Note

If you have installed only a Cisco AVS 3120 Application Velocity System, reporting functions are not available. You must be running the Management Console on a Cisco AVS 3180 Management Station in order to see the Report items in the Management Console. This note does not apply to users who have performed software upgrades to AVS 5.0 from older software products or on the Velocity Appliance.

AppScreen Web Application Firewall

The AppScreen Web Application Firewall enables the application appliance to provide web application security and intrusion protection. AppScreen is highly configurable, and the following types of request filtering are pre-configured:

- Binary blocking
- Cross-site script blocking
- Directory traversal blocking
- SQL injection blocking
- File upload blocking

AppScreen uses full content inspection and can scan and analyze all HTTP/HTTPS requests, preventing unwanted requests from going to the origin server.

AppScreen uses simple, policy-driven, rule-based management. AppScreen uses XML to allow administrators to set actions and notifications upon rule matches, at both the global and application class level, including blacklist and whitelist behaviors. AppScreen even provides additional policies (such as binary blocking) to supplement all standard rule matching.

AppScreen provides graphical views of incidents by severity. It also includes SNMP interfaces allowing alerts published to enterprise management systems.

For details on configuring AppScreen, see [Chapter 6, “AppScreen Configuration.”](#)

Operation

This section describes how the application appliance works and includes the following topics:

- [Detailed AVS-Client Interaction, page 2-19](#)
- [FlashForward Operation, page 2-23](#)

Detailed AVS-Client Interaction

This section describes the interaction of the application appliance 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 all-user condensation.

Visit One—New Client

The process begins with the client’s first visit to the web page since the application appliance has been deployed.

In this visit, the application appliance 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 application appliance also embeds some JavaScript code into the returned page (for details, see the [“Visit One JavaScript Example” section on page 2-19](#)). This code attempts to create an application appliance cookie on the client’s system. If it is successful, the application appliance knows that the client actually does support JavaScript and it is enabled. The presence of this cookie tells the application appliance that when this client requests this page (or a page from this class) in the future, the application appliance can return condensed content.

The application appliance may also gzip compress the returned page, if the client’s Accept-Encoding header indicates that it can receive gzipped responses.

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 application appliance cookie is shown at the top. The application appliance adds this JavaScript to the existing HTML page content.

```
<SCRIPT LANGUAGE="JavaScript">
//!--
document.cookie="FGNCDN=5.0-f98f1ec8-7b57-402f-b23b-77f708a9a26b;
path=//;expires=Sat, 17 Dec 2005 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"
1 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/il>
<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 application appliance determines if the client supports JavaScript by checking for the presence of the application appliance cookie that was created during the first visit. If it finds the cookie, it knows JavaScript is enabled and the client can handle delta pages. The application appliance then generates and delivers a delta page to the client. For details on the delta page contents, see the [“Visit Two JavaScript Example” section on page 2-20](#).



Note

The first delta page returned by the application appliance requires two requests from the client to the application appliance 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 application appliance, so it can be used to calculate future deltas. On subsequent visits, only the small delta pages are delivered, unless rebasing is required.

The application appliance may also gzip compress the returned page, if the client's Accept-Encoding header indicates that it can receive gzipped responses. Both delta and base pages are gzipped if the client can receive compressed pages.

On each subsequent visit, the application appliance always checks for the presence of the application appliance cookie. If the application appliance cookie is not present, the application appliance 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 application appliance—the delta file returned by the application appliance is not useful. Without the base page, the browser would not be able to access the content delivered by the application appliance because the client would attempt to apply application appliance deltas to a nonexistent base page. To handle this case, the application appliance 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 isFGNBaseCorrect=false;</script>
<script type="text/javascript"
src="/4grv3hs41d2bkt4wj41kcotmlh/986848530/ausr/_fgn_http_/www.foo.com/">
</script>
<noscript><META HTTP-EQUIV="Refresh" CONTENT="0;
URL=http://www.foo.com/?fineground=removeCookie">
Please click on <a href="http://www.foo.com/?fineground=removeCookie">
```

```

this url</a> if the page is not refreshed automatically in a few seconds
</noscript>
<script type="text/javascript">
if (!isFGNBaseCorrect)
    document.location.reload(true);
</script>
<script type="text/javascript" >
/*
(c) FineGround Networks, 2000-2003. All rights reserved, patent pending V 9.0.0
Build Date Aug 20 2005
Build Time 19:21:47
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>

```

Let us look at each section of this file in detail:

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

This code defines the default FGNBaseCorrect 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 application appliance'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 application appliance node). Without this feature, the browser cannot access the content delivered by the application appliance because the client would attempt to apply application appliance deltas to a nonexistent base page. This feature addresses this problem through JavaScript code within the application appliance 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 application appliance.

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

This code refers the client to the base file for the requested content. The base file can be retrieved from a network cache or the application appliance 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 file name is not the same as the originally requested page. As a result, base files are retrieved only by clients that utilize a Cisco AVS device.

Let's examine the base file naming convention in this example. The first string, "4grv3hs41d2bkt4wj41kcotmlh," is the application appliance ID that uniquely identifies the application appliance that generated this base file. It is a one-way hash based on the MAC address, IP address, and port of the application appliance node. The second string, "986848530," represents a modification timestamp of the base file that enables the application appliance to detect base file version changes.

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

This code enables the automatic cookie expiration feature, which allows the application appliance to guarantee 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 upon retrieving condensed content (JavaScript) from the application appliance. The solution is to include this <NOSCRIPT> tag in application appliance responses to clients. If JavaScript is disabled on the browser after the initial JavaScript-enabled request, client execution of this code will automatically force the client to expire the application appliance cookie and fetch subsequent pages uncondensed (without application appliance-generated JavaScript coding).

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

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

```
<script type="text/javascript">
/*
(c) FineGround Networks, 2000-2003. All rights reserved, patent pending V 9.0.0
Build Date Aug 20 2005
Build Time 19:21:47
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 code represents the content delta information. Using simple string-manipulating JavaScript functions defined in the base file, it enables the client to construct the newly requested page from the previously retrieved base file.

Cache Control Headers

The delta pages sent to clients by the application appliance use exactly the same HTTP cache control headers as the original page served by the origin server. This allows network caches to cache these pages in the same manner as the original page.

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

FlashForward Operation

These sections describe how FlashForward works:

- [FlashForward Overview, page 2-23](#)
- [FlashForward and Delta Optimization Options, page 2-23](#)
- [FlashForward and Repeat Visits, page 2-23](#)
- [How FlashForward Works With Delta Optimization, page 2-24](#)
- [How FlashForward Works Without Delta Optimization, page 2-27](#)
- [How FlashForward Works With CDN URLs, page 2-29](#)

FlashForward Overview

If an embedded object has a “Last-Modified” date and an “Expires” date so that the object has not yet expired in the browser cache, a revisit in a new browser session to the HTML page that references the object will not trigger any kind of HTTP GET request for the object. This feature reduces upstream HTTP request traffic and accelerates the delivery of the page.

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 to configure FlashForward by itself or delta optimization by itself.

Configuring FlashForward by itself (without delta optimization) may be appropriate for sites with small HTML pages that contain many cacheable embedded objects. Such a configuration will not provide HTML acceleration benefits because it will not accelerate HTML delivery in repeat visits. It will, however, accelerate embedded object delivery in repeat visits, typically across browser sessions. A side benefit of this type of configuration is that it eliminates the requirement that client browsers support cookies or JavaScript-based DHTML. FlashForward by itself supports all browsers automatically, but delta optimization requires the browser to support DHTML.

Configuring delta optimization without FlashForward may be appropriate for sites with large HTML pages that contain few cacheable embedded objects. Such a configuration will accelerate HTML delivery both within and across browser sessions but will not accelerate embedded object delivery across browser sessions. Any application appliance configuration that uses delta optimization requires that the client browser support both cookies and JavaScript-based DHTML. We recommend that you use both delta optimization and FlashForward for optimal results.

FlashForward and Repeat Visits

Most users will first experience FlashForward acceleration benefits on their first repeat visit to a given page. The examples in this section assume that the application appliance was just installed and include an extra initial step required to prime the application appliance cache with FlashForwarded objects. The examples show that the user will start to see FlashForward benefits on the second repeat visit to a given page (the third overall visit). Once the application appliance cache is primed, clients will begin to see FlashForward benefits on their first repeat visit (the second overall visit).

**Note**

Most users will experience FlashForward benefits on their first repeat visit to a given page. (This “First Repeat Visit” represents the actual first visit for clients after the application appliance cache is primed.) Regardless of whether FlashForward is configured 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 application appliance 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 take advantage of the first client’s sacrifice in priming the application appliance cache and will gain the FlashForward benefits on the first repeat visit and later. Why? 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 application appliance 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 application appliance that has just been installed (its cache is empty). No base files or objects are yet in the application appliance cache. The very first request for a given URL overall (not the first visit per client) is used to prime the application appliance cache as follows:

1. The client requests a URL. The application appliance proxies it to the origin server.
2. The origin server delivers the HTML page to the application appliance.
3. The application appliance retrieves the page, parses through the HTML 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 application appliance because for this first visit, the application appliance cache has not yet been primed with the embedded objects. The application appliance prepends the application appliance cookie through JavaScript and delivers the page compressed but otherwise unaltered (a standard cookie-drop page is created and delivered without any FlashForwarding taking place).
4. The application appliance creates the base page as usual because this visit is the first to the URL by anyone.
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 application appliance is a proxy, the HTTP GETs are passed through the application appliance to the origin server and the subsequent object responses (HTTP “200 OK”) from the origin server are passed through the application appliance to the client. The application appliance caches all cacheable objects as they pass through, which primes the cache. In this way, the application appliance uses the client’s HTTP GET requests to populate the cache rather than resorting to some complex cache pre-population capability. This process only occurs the first time new objects are delivered from the server to the client (only when the application appliance determines that the original HTML references objects that are not in the cache at the time of the HTML request). The application appliance 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 application appliance sees its cookie and knows that the client can support optimized responses.
3. The application appliance proxies the request to the origin server.
4. The origin server creates and delivers the HTML page to the application appliance.
5. The application appliance parses through the HTML looking for references to embedded objects and checks if the referenced objects are cached locally. If so, the application appliance 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 application appliance issues HTTP IMS requests to the origin server to determine whether or not the cached objects are still valid.



Note

The application appliance does not issue 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, it uses the application appliance's cache freshness settings to determine how often it should issue IMS requests. For example, if you configure the cache expiration setting to be 10 minutes, the application appliance 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 application appliance 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 application appliance 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 application appliance 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 application appliance 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 application appliance cache) rather than the original objects on the origin server.
 7. The application appliance 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 application appliance compresses and delivers the delta page to the client.
 9. The client retrieves the delta page and reconstructs the rewritten HTML from it (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 application appliance.
11. The client requests the FlashForwarded objects from the application appliance; the application appliance 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 application appliance sees its cookie and knows that the client can support condensed responses.
3. The application appliance proxies the request to the origin server.
4. The origin server creates and delivers the HTML page to the application appliance.
5. The application appliance parses through the HTML looking for references to embedded objects and checks if the referenced objects are cached locally. If so, the application appliance checks if the cached objects can be FlashForwarded (determines if the application appliance is configured to FlashForward the object type). If the objects are both cacheable and can be FlashForwarded, the application appliance 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 application appliance 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 application appliance 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 application appliance 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 application appliance 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 application appliance compresses and delivers the delta page to the client.
9. The client retrieves the delta page and reconstructs the rewritten HTML from it (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 application appliance.
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 application appliance 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 application appliance 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 application appliance 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 application appliance cache as follows:

1. The client requests a URL. The application appliance proxies it to the origin server.
2. The origin server delivers the HTML page to the application appliance.
3. The application appliance retrieves the page, parses through the HTML 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 application appliance because the application appliance cache has not yet been primed with the embedded objects. The application appliance delivers the page compressed but otherwise unaltered as usual (a standard cookie-drop page is created and delivered without any FlashForwarding 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 looking for references to embedded objects, and requests the embedded objects directly from the origin server through HTTP GET requests.
5. Because the application appliance is a proxy, the HTTP GETs are passed through the application appliance to the origin server and the subsequent object responses (HTTP “200 OK”) from the origin server are passed through the application appliance to the client. The application appliance caches all cacheable objects as they pass through, thus priming the cache. The application appliance 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 only occurs the first time that new objects are delivered from the server to the client (when the original HTML references objects not yet in the cache at the time of the HTML request). The application appliance 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 application appliance proxies the request to the origin server.
3. The origin server creates and delivers the HTML page to the application appliance.

4. The application appliance parses through the HTML looking for references to embedded objects and checks if the referenced objects are cached locally. If so, the application appliance 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 application appliance 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 application appliance 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 application appliance 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 application appliance 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 application appliance 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 application appliance 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 application appliance.
8. The client requests the FlashForwarded objects from the application appliance; the application appliance 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 application appliance proxies the request to origin server.
3. The origin server creates and delivers the HTML page to the application appliance.
4. The application appliance parses through the HTML looking for references to embedded objects and checks if the referenced objects are cached locally. If so, the application appliance 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 application appliance 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 application appliance 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 application appliance 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 application appliance 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 application appliance 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 application appliance.
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 application appliance 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 application appliance to still be valid.

Embedded Objects Referenced Within JavaScript and Not Within HTML

To find references to embedded objects, the application appliance parses for `img`, `script`, `href`, and `background` tags within the HTML. It will not find references to embedded objects within JavaScript. Because the application appliance caches all cacheable embedded objects during the priming stage, it will still FlashForward cacheable JavaScript-embedded objects by adding an Expires header to them without renaming them. The date/time associated with the added Expires header is defined by the application appliance cache freshness settings (CacheMinTTL, CacheTTLPercent, and CacheMaxTTL). This FlashForward JavaScript-embedded objects feature is off by default and must be enabled explicitly through the ExpiresSetting parameter within the `fgn.conf` configuration file.

How FlashForward Works With CDN URLs

This feature enables the application appliance to apply FlashForward object acceleration to objects and URLs transformed by content delivery networks (CDNs).

As an example to clarify the problem, 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
```

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.

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 cases, the CDN-modified URLs consist of a CDN ID portion, followed by the original URL. To make this easier to see, 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 application appliance is able to extract the original portion of the URLs needed for FlashForward object validation.

This feature enables you to specify whether Akamai and/or Speedera are being used on the content being FlashForwarded. The feature enables the application appliance to identify and parse through such CDN URLs to extract the original URL portion and enables the application appliance to perform embedded object validation requests with the origin server as required by FlashForward. FlashForward transformation occurs as usual; for example, the application appliance 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_FineGround_vtmmi14xg2fvmlkxsxuk0ty1xd_FGN_V01.gif
```

Here tab_0.gif has been replaced with the FlashForward-transformed object name:

```
tab_0_FineGround_vtmmi14xg2fvmlkxsxuk0ty1xd_FGN_V01.gif
```

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

```
http://gateway.speedera.net/www.gateway.com/images
/cp/banners/homepage_bnr_broadband01_FineGround_5f1fdnjgsaigblyav4f42wmb1g_FGN_V01.gif
```

Here homepage_bnr_broadband01.gif has been replaced with the FlashForward-transformed object name:

```
homepage_bnr_broadband01_FineGround_5f1fdnjgsaigblyav4f42wmb1g_FGN_V01.gif
```

This feature enables the application appliance to FlashForward objects transformed by Akamai and Speedera and thus enables 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 configure an Application Class to modify the cache key, use the CacheKeyModifier directive, which is identical to the canonical URL definition in use. It is based on regular expressions. Here is an example:

```
<ApplicationClass AkamaiModifierClass>
  Url "^.*akamai.net.*www(.*\.)$"
  Url "^.*akamai.net.*www(.*\.)$"
  Url "^.*akamai.net.*www(.*\.)$"
  CacheKeyModifier http://www$(1)
  OptimizationPolicy NoDeltaOptimize,NoCompress,FlashForwardObject
  RequestCachePolicy OverrideAll
</ApplicationClass>
```

The AkamaiModifierClass matches images that have URLs transformed by Akamai (see the first three lines of the class definition). The parentheses in this example define a subexpression, that when matched, is used in the CacheKeyModifier line. Each “()” expression is numbered (index starting at 1) and can be used in any way using the expression $\$(number)$. For more details on using subexpressions, see [Table 5-4 on page 5-16](#).



Note

The application appliance requires a GNU POSIX regular expression syntax. For more information, see [Appendix F, “Regular Expressions.”](#)

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 application appliance for a considerable period of time, in which case FlashForward is not effective.

A similar example for Speedera is as follows:

```
<ApplicationClass SpeederaModifierClass>
  Url "^.*speedera.net.*www(.*\.gif)$"
  CacheKeyModifier http://www$(1)
  OptimizationPolicy NoDeltaOptimize,NoCompress, FlashForwardObject
  RequestCachePolicy OverrideAll
</ApplicationClass>
```

Essentially this is the same as the Akamai example, except the URLs that match this class are different.

