



Configuring Caching

This chapter provides an overview of the CSS caching feature and describes how to configure it for operation. Information in this chapter applies to all CSS models, except where noted.

The chapter includes the following major sections:

- [Caching Overview](#)
- [Caching Configuration Quick Start](#)
- [Configuring Caching Using Content Rules](#)
- [Configuring Network Address Translation Peering](#)

Caching Overview

Increasing demand for information on the Internet causes congestion and long delays in retrieving information. Because much of the same information is retrieved over and over again, saving and storing this information can satisfy subsequent requests with more efficiency and less bandwidth.

Saving and storing information locally is known as *caching*. With Web caching, copies of recently requested content are stored temporarily on a cache server in locations that are topologically closer to the client. The content is then readily available to be reused for subsequent client requests for the same content.

By storing content locally, you:

- Optimize network resources
- Conserve network bandwidth
- Reduce Internet congestion
- Improve network response time and overall service quality

Content Caching

You can make Web caching cost-effective and more reliable by deploying content caching in your network. By creating content rules to utilize your cache servers, the CSS acts as a cache front-end device by:

- Examining network traffic for Web content requests
- Bypassing the cache automatically for non-cacheable content
- Distributing content requests to maximize cache hits on services
- Bypassing the cache or redistributing content requests among the remaining cache services if a cache service fails

When a client requests content, the CSS:

- Intercepts the request for content
- Applies content intelligence by parsing the HTTP request header to distribute content requests to the cache servers

The CSS then either:

- Directs the request to the appropriate cache based on the load-balancing method you specify in the content rule (for example, destination IP address)
- Bypasses the cache servers and forwards the request to the origin server if the content is noncacheable

When the CSS directs the request to the cache server, the cache server either returns the requested content (if it has a local copy) or sends a new request for the content through the CSS to the origin server hosting the content. When the cache sends a new request for content and receives a reply from the origin server, it returns the response to the client. If the content is cacheable, the cache saves a copy of the content for future requests.

When the requested content is found on a local cache server, the request is known as a *cache hit*. When the requested content is not local and the cache initiates a new request for the content, the request is known as a *cache miss*.

The following sections provide CSS content caching examples:

- [Using Proxy Caching](#)
- [Using Reverse Proxy Caching](#)
- [Using Transparent Caching](#)
- [Using Cache Clustering](#)

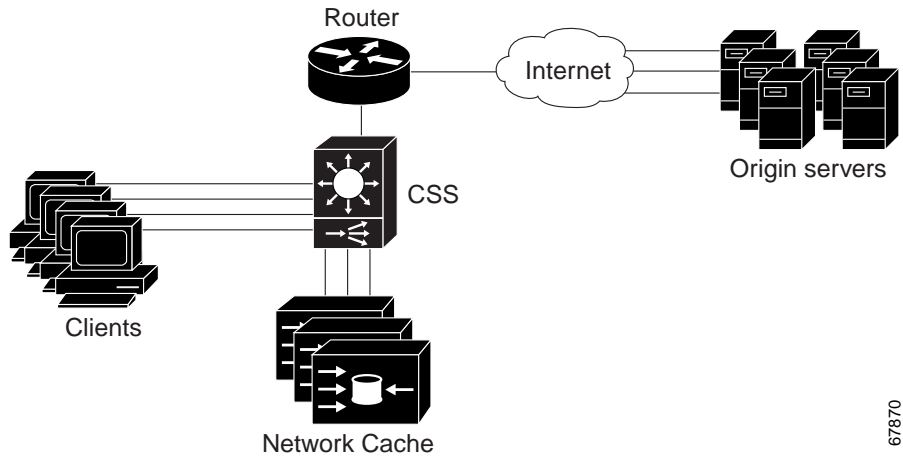
Using Proxy Caching

With proxy caching, each client is configured with the IP address of the proxy cache to which clients send content requests. You may also configure a URL for browsers to identify the location of the proxy configuration file for automatic proxy configuration. Each client's content request is sent directly to the proxy cache IP address. The cache returns the requested content if it has a local copy, or else it sends a new request to the origin server for the information.

If all cache servers are unavailable in a proxy cache configuration, the client request does not pass to the origin server because clients are configured with the proxy cache VIP.

Figure 7-1 shows an example of using a CSS in a proxy cache configuration.

Figure 7-1 Proxy Cache Configuration Example



67870

Using Reverse Proxy Caching

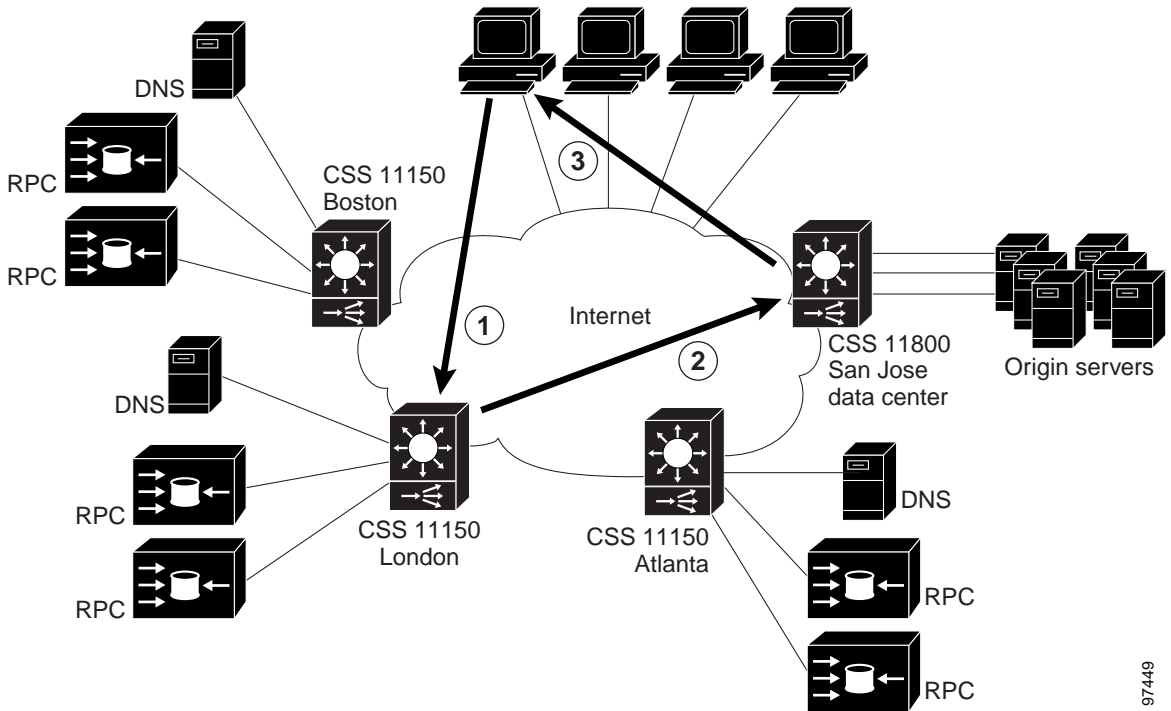
In a reverse proxy cache configuration, the proxy server is configured with an Internet-routable IP address. Clients are directed to the proxy server based on a Domain Name System (DNS) resolution of a domain name. To a client, the reverse proxy server appears like a Web server.

In a regular proxy cache configuration, the proxy server acts as a proxy for the client. In the reverse proxy configuration, the reverse proxy server acts as a proxy for the server. Also, a reverse proxy cache caches specific content, whereas proxy and transparent caches cache frequently requested content. Reverse proxy caches serve two primary functions:

- Replication of content to geographically dispersed areas
- Replication of content for load balancing

Figure 7-2 shows an example of a CSS 11800 and CSS 11150s in a reverse proxy cache configuration.

Figure 7-2 Reverse Proxy Cache Configuration Example



97449

Using Transparent Caching

Transparent caching deploys cache servers that are transparent to the browsers. You do not have to configure browsers to point to a cache server. Cache servers duplicate and store inbound Internet data previously requested by clients.

When you configure transparent caching on the CSS, the CSS intercepts and redirects outbound client requests for Internet data to the cache servers on your network. The cache returns the requested content if it has a local copy, or else it sends a new request to the origin server for the information.

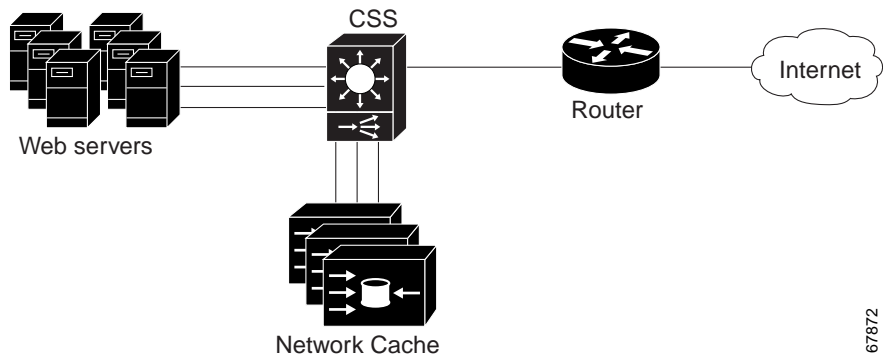
If all cache servers are unavailable in a transparent cache configuration, the CSS allows all client requests to progress to the origin servers.

A transparent caching configuration:

- Reduces network congestion caused by HTTP traffic
- Increases network efficiency
- Decreases the time required to fulfill a client request by accessing locally stored information rather than obtaining the same information across the Internet

Figure 7-3 shows an example of a typical transparent cache configuration.

Figure 7-3 *Transparent Cache Configuration Example*



67872

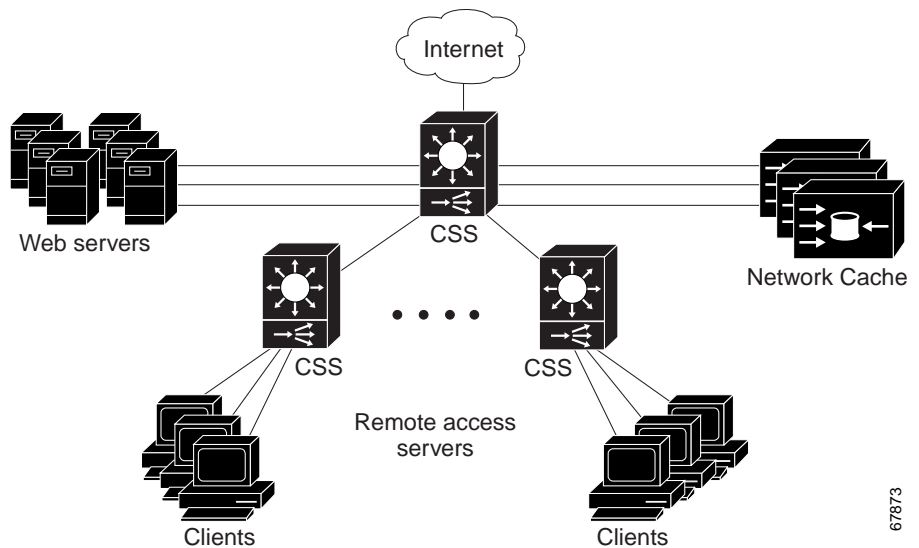
Using Cache Clustering

Multiple caches deployed at a single location is referred to as cache clustering. Cache clustering provides:

- Scalability
- Redundancy
- Transparency
- Simplified administration

Figure 7-4 shows an example of using content caching in a cache cluster configuration.

Figure 7-4 Cache Cluster Configuration Example



67873

Caching Configuration Quick Start

[Table 7-1](#) provides the steps to configure service `serv1` as a caching service. Each step includes the CLI command required to complete the task. Ensure that you have configured services, owners, and content rules prior to configuring CSS caching.



Note

When using content caching, the `keepalive` type must be `ICMP` (default setting).

For a complete description of each caching command, see the sections following [Table 7-1](#).

Table 7-1 *Caching Configuration Quick Start*

Task and Command Example

1. Specify a service type (**type local, type proxy-cache, type redirect, type transparent-cache**). The default is `local`.

```
(config-service[serv1])# type transparent-cache
```

2. Create an Extension Qualifier List (EQL) where you specify which content types the CSS caches.

```
(config)# eql graphics
```

```
(config-eql[graphics])#
```

3. Describe the EQL by entering a quoted text string with a maximum length of 63 characters.

```
(config-eql[graphics])# description "This EQL specifies cacheable graphic files"
```

4. Specify the extension for content you want the CSS to cache. Enter a text string from 1 to 8 characters.

```
(config-eql[graphics])# extension jpeg
```

Optionally, you may provide a description of the extension type. Enter a quoted text string with a maximum length of 64 characters.

```
(config-eql[graphics])# extension gif "This is a graphics file"
(config-eql[graphics])# exit
(config)#
```

Table 7-1 Caching Configuration Quick Start (continued)

Task and Command Example
5. Specify the EQL in a content rule to match all content requests with the desired extensions.
<pre>(config-owner-content [arrowpoint.com-rule1])# url "/*" eql graphics</pre>
6. Configure the load-balancing method for the cache content rule. The default is roundrobin.
<pre>(config-owner-content [arrowpoint.com-rule1])# balance domain</pre>
7. Specify a failover type to define how the CSS handles content requests when a service fails (bypass , next). The default is linear.
<pre>(config-owner-content [arrowpoint.com-rule1])# failover bypass</pre>
8. Display the EQL configuration.
<pre>(config-owner-content [arrowpoint.com-rule1])# show eql</pre>
9. Display the content rule to show the cache configuration.
<pre>(config-owner-content [arrowpoint.com-rule1])# show rule</pre>

Configuring Caching Using Content Rules

Configure caching using content rules. When you are creating caching content rules, the additional configuration requirements involve:

- Specifying a service type that supports caching
- Specifying a failover type for the cache servers
- Configuring a load-balancing algorithm that supports caching
- Configuring EQLs to identify file extensions that the CSS should direct to the cache services

**Note**

If you are running the Inktomi Traffic Server on a system that does not listen in promiscuous mode and want to bypass the Inktomi Adaptive Redirect module (that is, you want to send traffic directly to port 8080 instead of port 80), specify the CSS service type as **type proxy-cache**. Configuring the CSS service type to **type proxy-cache** causes the CSS to perform full Network Address Translation (NAT) when directing traffic to the Traffic Server.

Specifying a Service Type

The CSS enables you to specify the following cache-specific service types using the **type** command. The default service type is local.

- **type nci-direct-return** - Specifies the service as NAT Channel indication for direct return. Use with reverse proxy cache and NAT peering.
- **type nci-info-only** - Specifies the service as NAT Channel indication for information only. Use with reverse proxy cache and NAT peering.
- **type proxy-cache** - Specifies the service as a proxy cache. This option bypasses content rules for requests coming *from* the cache server. In this case, bypassing content rules prevents a loop between the cache and the CSS.
- **type rep-cache** - Specifies the service as a replication cache.
- **type rep-cache-redirect** - Specifies the service as a replication cache with redirect.
- **type transparent-cache** - Specifies the service as a transparent cache. No content rules are applied to requests from this service type. Bypassing content rules in this case prevents a loop between the cache and the CSS.

For example, to specify service `serv1` as a proxy cache, enter:

```
(config-service[serv1])# type proxy-cache
```

The CSS recognizes and forwards the following HTTP methods directly to the destination server in a transparent caching environment. However, the CSS does not load balance these methods.

- RFC 2068: OPTIONS, TRACE
- RFC 2518: PROPFIND, PROPPATCH, MKCOL, MOVE, LOCK, UNLOCK, COPY, DELETE

**Note**

To enable the CSS to redirect a request to a remote service when a request for content matches the rule, you must specify a URL for the content rule.

Specifying a Failover Type

To define how the CSS handles content requests when a cache service fails or is suspended, use the **failover** command. For the CSS to use this setting, ensure that you configure a keepalive for each service; that is, do not set the keepalive type to none (default keepalive is ICMP). The CSS uses the keepalive settings to monitor the cache services to determine server health and availability. See [Chapter 1, Configuring Services](#) for more information on the **keepalive** command.

By default, the CSS uses a linear failover method, which distributes the content requests to the failed service evenly among the remaining services.

**Note**

If you remove a service (using the **remove service** command) the CSS rebalances the remaining services. The CSS does not use the failover setting.

This command supports the following options:

- **failover bypass** - Bypass all failed services and send the content request directly to the origin server. This option is used in a proxy or transparent cache environment when you want to bypass the failed cache and send the content request directly to the server that contains the content.
- **failover linear** (default) - Distribute the content request evenly between the remaining services.
- **failover next** - Send the content requests to the cache service next to the failed service. The CSS selects the service to redirect content requests to by referring to the order in which you configured the services.

For example, enter:

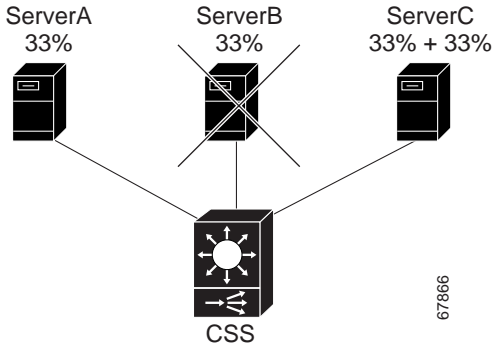
```
(config-owner-content [arrowpoint.com-rule1])# failover bypass
```

To restore the default failover method of linear, enter:

```
(config-owner-content [arrowpoint.com-rule1])# no failover
```

Figure 7-5 shows three cache services configured for failover **next**. If ServerB fails, the CSS sends ServerB content requests to ServerC, which was configured after ServerB in the content rule.

Figure 7-5 Cache Services Configured for Failover Next - Example 1



As shown in Figure 7-6, if ServerC fails, the CSS sends ServerC content requests to ServerA because no other services were configured after ServerC.

Figure 7-6 Cache Services Configured for Failover Next - Example 2

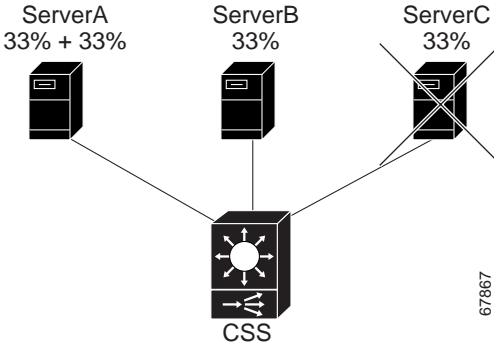


Figure 7-7 shows three cache services configured for **failover linear** (the default). If you suspend ServerB or if it fails, the CSS does not rebalance the services. It evenly distributes ServerB cache workload between servers A and C.

Note that Figure 7-7 and Figure 7-8 use the alphabet to illustrate division balance.

Figure 7-7 *Suspended or Failed Cache Service Configured for Failover Linear*

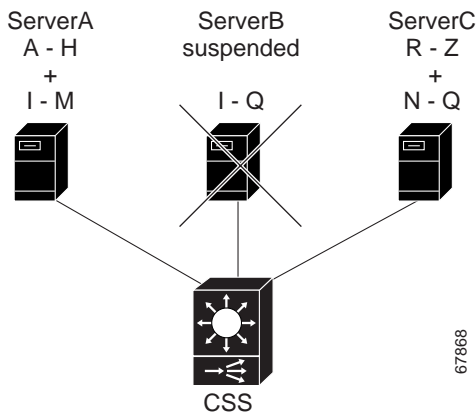
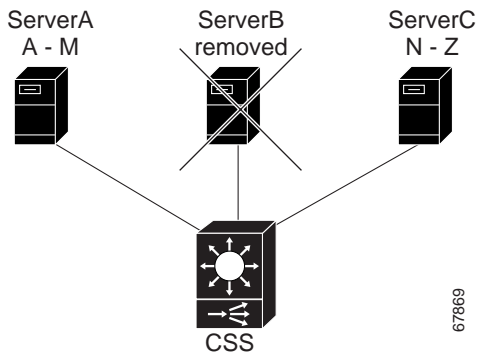


Figure 7-8 also shows three cache services configured for failover **linear**, but in this example, you remove ServerB using the **remove service** command from owner-content mode. Because the CSS does not apply the failover setting when you remove a service, it rebalances the remaining services.

Figure 7-8 *Removing a Cache Service Configured for Failover Linear*



Configuring Load Balancing

To specify the load-balancing algorithm for a content rule, use the **balance** command. This command is available in content configuration mode. The options are:

- **balance aca** - ArrowPoint Content Awareness load-balancing algorithm (see the “Using ArrowPoint Content Awareness Based on Server Load and Weight” section in [Chapter 1, Configuring Services](#)). ACA balances the traffic over the services based on load or on server weight and load.
- **balance destip** - Destination IP address division algorithm. The CSS directs all client requests with the same destination IP address to the same service. This option is typically used in a caching environment.
- **balance domain** - Domain name division algorithm. The CSS divides the alphabet evenly across the number of caches. It parses the host tag for the first four letters following the first dot and then uses these characters of the domain name to determine to which server it should forward the request. This option is typically used in a caching environment.
- **balance domainhash** - Internal CSS hash algorithm based on the domain string. The CSS parses the host tag and does an XOR hash across the entire host name. It then uses the XOR hash value to determine to which server to forward the request. This method guarantees that all requests with the same host tag will be sent to the same server in order to increase the probability of a cache hit. This option is typically used in a caching environment.



Note

If you are using the **domainhash** load-balancing method with proxy cache services, you may see duplicate sites across caches because the CSS balances on the first GET request in a persistent connection unless the subsequent GET request does not match a rule with the same proxy service specified. If you are concerned about duplicate hits across caches, reset persistence to remap and disable persistence on the rule. Issue the **(config) persistence reset remap** command globally and the **(config-owner-content) no persistent** command on the content rule.

- **balance leastconn** - Least connection algorithm. This balance method chooses a running service that has the least number of connections.

- **balance roundrobin** - Roundrobin algorithm (default). The CSS resolves the request by evenly distributing the load to resolve domain names among local and remote content domain sites.
- **balance srcip** - Source IP address division algorithm. The CSS directs all client requests coming from the same source IP address to the same service. This option is generally used in a caching configuration.
- **balance url** - URL division algorithm. The CSS divides the alphabet evenly across the number of caches. It then parses the URL for the first four characters located after the portion of the URL matched on by the rule. For example, if the URL in a content rule is configured for /news/*, the CSS will balance on the first four characters following /news/. This option is typically used in a caching environment.
- **balance weightedrr** - Weighted roundrobin algorithm. The CSS uses roundrobin but weighs some services more heavily than others depending on the server's configured weight. All servers have a default weight of 1. To set a server weight, use the **add service weight** command in owner-content mode.
- **balance urlhash** - Internal CSS hash algorithm based on the URL string. The CSS parses the URL and performs an XOR hash across the URL. It then uses the XOR hash value to determine to which server to forward the request. This method guarantees that all requests for the same URL will be sent to the same server in order to increase the probability of a cache hit. This option is typically used in a caching environment.

**Note**

A Layer 5 content rule supports the HTTP CONNECT, GET, HEAD, POST, PUSH, and PUT methods. The CSS recognizes and forwards the following HTTP methods directly to the destination server in a transparent caching environment. Note that the CSS does not load balance these HTTP methods. RFC 2068: OPTIONS, TRACE; RFC 2518: PROPFIND, PROPPATCH, MKCOL, MOVE, LOCK, UNLOCK, COPY, DELETE.

In a transparent caching environment (for example, no VIP address on a Layer 5 content rule), the CSS bypasses these HTTP methods, and they are forwarded to the destination server.

For example, to specify weighted roundrobin algorithm load balancing, enter:

```
(config-owner-content [arrowpoint-rule1]) # balance weightedrr
```

To revert the balance type to the default of roundrobin, enter:

```
(config-owner-content [arrowpoint-rule1]) # no balance
```

Configuring a Double-Wildcard Caching Content Rule

When you want to optimize Layer 3 and Layer 4 TCP/IP traffic, configure a content rule for transparent caching without specifying the VIP address and port number. This configuration may be particularly useful in a wireless environment where there is intelligence built into the backend server.

If all other matching criteria in the content rule are met by the client request, a request with any VIP or port will match the rule. This is called a double-wildcard caching rule. You still need to specify the protocol in the rule. Typically, use this type of rule when you are load-balancing services of **type transparent-cache**. However, you can configure this type of rule with other service types as well.



Note

If you have a configuration that requires a double-wildcard rule, be aware that the client request will match on this rule when the client attempts to connect directly to a server IP address.

Enabling Content Requests to Bypass Caches

The following sections describe how to enable content requests to bypass caches:

- [Using the param-bypass Command](#)
- [Using the cache-bypass Command](#)
- [Using the bypass-hosttag Command](#)

Using the param-bypass Command

Use the **param-bypass** command to enable content requests to bypass transparent caches when the CSS detects special terminators in the requests. The terminators “#” and “?” indicate that the content is dependent on the arguments that follow the terminators. Because the content returned by the server is dependent on the content request itself, the returned content is not cacheable.

This command contains the following options:

- **param-bypass disable** (default) - Content requests with special terminators do not bypass transparent caches.
- **param-bypass enable** - Content requests with special terminators bypass transparent caches and are forwarded to the origin server.

For example, to enable the **param-bypass** command, enter:

```
(config-owner-content [arrowpoint-rule1])# param-bypass enable
```

Using the cache-bypass Command

By default, a CSS does not apply content rules to requests from a proxy or transparent-cache type service going to the origin server when the cache does not contain the requested content. Use the **no cache-bypass** command to allow the application of content rules to requests originating from a proxy or transparent cache. Use the **cache-bypass** command to restore the default behavior of the CSS after you have issued the **no cache-bypass** command.

For example, to allow the CSS to apply content rules to requests from a proxy or transparent-cache type service, enter:

```
(config-service [serv1])# no cache-bypass
```

To restore the CSS default behavior after issuing the **no cache-bypass** command, enter:

```
(config-service [serv1])# cache-bypass
```

Using the `bypass-hosttag` Command

Use the **`bypass-hosttag`** command to allow a CSS configured as a Client Side Accelerator (CSA) to bypass a cache farm and establish a connection with the origin server to retrieve noncacheable content. The domain name from the `host-tag` field is used to look up the origin IP address on the CSA.

**Note**

Use the **`bypass-hosttag`** command only with a CSS operating in a CSA environment. For details on CSA, refer to the *Cisco Content Services Switch Advanced Configuration Guide*.

For example, enter:

```
(config-service[serv1])# bypass-hosttag
```

To disable bypassing cache for noncacheable content, enter:

```
(config-service[serv1])# no bypass-hosttag
```

Configuring Network Address Translation for Transparent Caches

Use the **`transparent-hosttag`** command to enable destination Network Address Translation (NAT) for the transparent cache service type. This command NATs the destination address of the client's packet (forwarded by the CSS to the cache) to the origin server IP address for the requested domain. Using this command ensures that the cache always has the current origin server IP address based on periodic DNS lookups that the CSS performs for all accelerated domains.

The alternative is to manually configure all origin server IP addresses on the cache, which may or may not support static configuration. Also, statically configured IP addresses can become obsolete if the origin server IP address changes. For caches that support DNS resolution and use the DNS response to fetch content or that support configuration of origin server IP addresses, **`transparent-hosttag`** is not required but is recommended.

**Note**

You can use the **transparent-hosttag** command only with a CSS operating in a Client Side Accelerator (CSA) environment. For details on CSA, refer to the *Cisco Content Service Switch Advanced Configuration Guide*.

For example, enter:

```
(config-service[ serv1])# transparent-hosttag
```

To disable destination NATing for the transparent cache service type, enter:

```
(config-service[ serv1])# no transparent-hosttag
```

Configuring Network Address Translation Peering

NAT peering allows clients to connect to remote Web sites through CSSs and have the return traffic use the shortest network path back to the client. The forward path from the client to the server is through TCP connections between two CSSs, but the reverse path from the server to the client may take the shortest network route rather than traversing back through the CSSs.

**Note**

NAT peering is part of the CSS Enhanced feature set.

NAT peering allows the CSS to:

- Forward client connections to a remote CSS
- Perform the final translation at the remote CSS, which allows return traffic packets to flow to the client through any network path
- Preserve the client IP address when forwarding traffic to the origin server

To perform NAT transformations on a TCP flow, the client-side CSS forwards traffic to the server-side CSS through a NAT channel. This channel uses a special TCP option called the NAT Channel Indication (NCI) option. This option indicates to the server-side CSS that NAT parameters are in use, and contains the original source and destination IP addresses, and TCP port numbers. This option also has a spoof bit to indicate that part of the flow has been spoofed and the rest of the forward path must be established before the destination CSS can use the information in the packet to perform the NAT transformations for the reverse path.

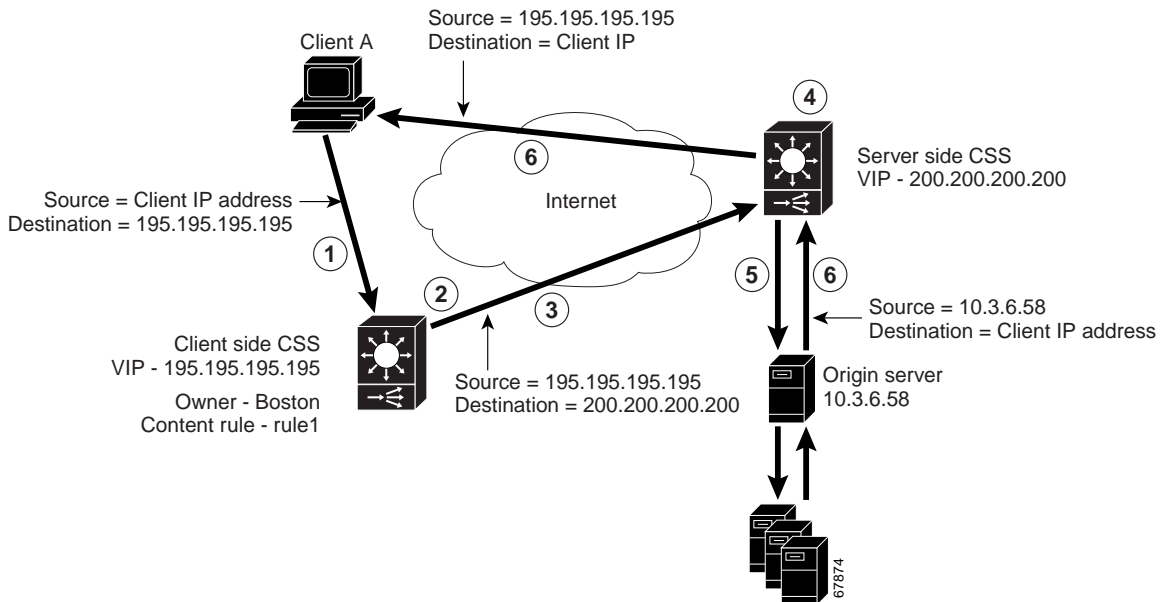
**Note**

Spoofing occurs when a CSS requires information from the HTTP request, (such as host tag, filename, file extension) in order to make a load-balancing decision.

The server-side CSS preserves the client address and port. This allows the origin server to maintain statistics based on the original traffic source addressing data, and allows the return path to be independent of the forwarding path.

Figure 7-9 shows an example of NAT peering. The steps that follow describe this example.

Figure 7-9 NAT Peering Configuration Example



1. Client A sends a content request for /bostonInfo.html from the client-side CSS (CSS1, VIP 195.195.195.195).
2. The client-side CSS matches the request to its content rule, which specifies a service located on the server-side CSS (CSS2, VIP2 200.200.200.200). The server-side CSS service is configured for service type **nci-direct-return**. This service type informs the client-side CSS to include the NCI option in the TCP packet sent to server-side CSS. If a Layer 5 rule is matched, the spoof bit in the NCI option is set.
3. The client-side CSS sends the TCP packet to the server-side CSS. Source address group mapping maps the Client A source address and port to those from the client-side CSS. The TCP packet contains the client-side CSS source information, the server-side CSS destination information, and the original source and destination information from Client A.
4. The server-side CSS determines whether the spoof bit has been set in the packet. If the bit is set, the CSS stores the NAT information until the connection is spoofed. The server-side CSS sets up the forward and return paths. The server-side CSS then matches the request from the client-side CSS on a content rule.



Note The server-side CSS (in Figure 7-9) would use the NCI option in a packet if the VIP rule is directed at a local, proxy-cache, or transparent cache service.

5. The server-side CSS sends the request to the origin server with the destination IP address translated to the origin server IP address and the source IP address translated to the client IP address.
6. The origin server responds directly back to Client A. As the packet flows through the server-side CSS, that CSS translates the source IP address to the CSS1 VIP. The destination IP address is the client IP address.

Configuring NAT Peering

All NAT peering configuration occurs on the client-side CSS. During the configuration consider the following:

- When you configure the NCI service as **nci-direct-return**, the service must be directed to the VIP on the server-side CSS to indicate an endpoint for the connection. The server-side CSS always uses the **nci-direct-return** option to modify the source address and port that the server sees. When the **nci-direct-return** service is used on the client-side, the return path is modified to directly return to the client.
- When you are specifying an NCI service type, you must specify:
 - **type nci-direct-return** to represent a VIP on another CSS
 - **type nci-info-only** for any Web server

[Table 7-2](#) describes the steps necessary to configure NAT peering using command examples based on the configuration in [Figure 7-9](#). Because NAT peering applies to Layer 3 as well as Layer 5 rules, the port, protocol, and URL rule examples shown in [Table 7-2](#) are optional.

Table 7-2 NAT Configuration Quick Start

Task and Command Example

1. On the client-side CSS (CSS1), create content rules to configure the server-side CSS (CSS2) as a service.

- a. Create service CSS2.

```
CSS1 (config)# service CSS2
```

- b. Configure CSS2 VIP as the service IP address.

```
CSS1 (config-service[CSS2])# ip address 200.200.200.200
```

- c. Configure CSS2 as a service type **nci-direct-return**.

```
CSS1 (config-service[CSS2])# type nci-direct-return
```

- d. Activate the content rule.

```
CSS1 (config-service[CSS2])# active
```

Table 7-2 NAT Configuration Quick Start (continued)

Task and Command Example

2. On the client-side CSS (CSS1), create content rules with the criteria required for the client-side CSS (CSS1) to forward traffic to the server-side CSS (CSS2).

- a. Create an owner.

```
CSS1 (config)# owner boston.com
```

- b. Name the content rule and assign it the owner.

```
CSS1 (config-owner[boston.com])# content rule1
```

- c. Configure the CSS1 VIP.

```
CSS1 (config-owner-content[boston.com-rule1])# vip  
195.195.195.195
```

- d. Configure port and protocol.

```
CSS1 (config-owner-content[boston.com-rule1])# port 80  
CSS1 (config-owner-content[boston.com-rule1])# protocol tcp
```

- e. Define the URL.

```
CSS1 (config-owner-content[boston.com-rule1])# url  
"//bostoninfo.html/"
```

- f. Add CSS2 as the service.

```
CSS1 (config-owner-content[boston.com-rule1])# service CSS2
```

- g. Activate the rule.

```
CSS1 (config-owner-content[boston.com-rule1])# active
```

Table 7-2 NAT Configuration Quick Start (continued)

Task and Command Example

3. On the client-side CSS (CSS1), create a source group for the client traffic. CSS1 will translate the Client A IP address to the IP address defined in the source group. To configure a source group:

- a. Create the source group.

```
CSS1 (config)# group boston
CSS1 (config-group[boston])#
```

- b. Define the CSS1 VIP as the IP address into which the Client A IP address will be translated.

```
CSS1 (config-group[boston])# vip 195.195.195.195
```

- c. Activate the source group.

```
CSS1 (config-group[boston])# active
```

4. On the client-side CSS (CSS1), create an access control list (ACL) clause to specify which source IP addresses use the source group. Note that clause 20 is a required clause that permits all other traffic. Without clause 20, all traffic not defined in clause 10 is denied.

```
CSS1 (config)# acl 1
CSS1 (config-acl[1])# clause 10 permit tcp any destination
content boston.com/rule1 sourcegroup boston
CSS1 (config-acl[1])# clause 20 permit any any destination
any apply circuit-(VLAN1)
```

5. On the server-side CSS (CSS2), configure the origin server connected to CSS2.

- a. Create origin server serv1.

```
CSS2 (config)# service serv1
```

- b. Configure an IP address for serv1.

```
CSS2 (config-service[serv1])# ip address 10.3.6.58
```

- c. Activate the server.

```
CSS2 (config-service[serv1])# active
```

Table 7-2 NAT Configuration Quick Start (continued)

Task and Command Example

6. On the server-side CSS (CSS2), configure content rules with the criteria required to forward content requests to `serv1`.
 - a. Create an owner.

```
CSS2 (config)# owner boston.com
```
 - b. Name the content rule and assign it the owner.

```
CSS2 (config-owner[boston.com])# content rule1
```
 - c. Configure the CSS2 VIP.

```
CSS2 (config-owner-content[boston.com-rule1])# vip
200.200.200.200
```
 - d. Configure port and protocol.

```
CSS2 (config-owner-content[boston.com-rule1])# port 80
CSS2 (config-owner-content[boston.com-rule1])# protocol tcp
```
 - e. Add `serv1` as the service.

```
CSS2 (config-owner-content[boston.com-rule1])# service serv1
```
 - f. Define a URL.

```
CSS2 (config-owner-content[boston.com-rule1])# url "/"
```
 - g. Activate the rule.

```
CSS2 (config-owner-content[boston.com-rule1])# active
```
-

