Abstract

This paper proposes translucent caching as an alternative to transparent caches. Translucent caches use the fact that network routing forwards a request for an object along the best path from the client to the object’s home server. Along this best path, routers direct the request toward a cache chosen among a collection of nearby translucent caches. Unlike transparent caching, which relies on routers to serve as TCP connection intermediaries, translucent caches only use routers to get next-hop cache information. By doing away with TCP connection intermediaries, translucent caches conserve the end-to-end argument-based robustness.

1 Introduction

This paper discusses ways of scaling Web caches. It proposes translucent caching, an alternative to transparent caching. Translucent caching works by directing a request for an object toward caches along the way from the client to the object’s origin server. It relies on the network routing fabric to route the request along the shortest path between the source and destination. Translucent routers along the way intercept the request and provide the address of a next-hop cache server to the previous hop. The previous-hop cache then sends the request to the indicated next-hop cache. Similarly to transparent caching, translucent routers balance load by directing a request to a cache chosen from a collection of next-hop caches. Unlike transparent caching, which relies on routers to serve as TCP connection intermediaries, translucent caches only use routers to get next-hop cache information. By establishing the end-to-end TCP connections themselves, translucent caches avoid the problems caused by route flapping.

For performance, translucent caches can cache next-hop information to avoid the extra round-trip time. In our design, we also include a maximum number of intercepts option. By setting this flag, we limit the number of times a request gets intercepted by translucent caches along its way to the object’s origin server.

The paper is organized as follows. In Section 2 we review several load balancing schemes; they work by partitioning web traffic among centrally administered caches, such as caches within an ISP or Internet backbone connectivity provider. We single out transparent caching as a router-supported load balancing mechanism and discuss its advantages and disadvantages. Section 3 introduces translucent caches and present our design and implementation. It concludes by presenting a simple clustering failover scheme we designed and implemented to improve translucent caching availability and fault tolerance.
2 Load Balanced Caches

One way to make Web caching scale is to partition traffic among a group of collaborating caches, including proxy caches, network service provider, and content provider caches. In this section we review transparent caching – a router-supported load balancing mechanism, and discuss its strengths and drawbacks.

We start by reviewing some software-based load balancing schemes, namely proxy auto-configuration and server-side re-direction.

2.1 Software-based Load Balancing

Proxy Auto-Configuration

Most available browsers support proxy auto-configuration. It works by configuring browsers with the URL of a configuration program, which maps URLs to specific proxy servers. Configuration programs typically use a hash function based on the URL’s hostname to perform the URL-proxy mapping. Hostname hash functions avoid duplicate caching and maximize cache hits by consistently mapping requests to an object to the same cache server.

The problem with proxy auto-configuration is that it is not transparent to the user. Since it involves browser configuration, users need to get involved in downloading new browsers or re-configuring the existing one every time new cache deployment or configuration is performed. This often leads to customer support calls and user discontent.

Server-Side Re-direction

Cache server load balancing solutions can complement and are orthogonal to client-side mechanisms. Resonate’s Dispatch [9] is an example of a server-side software-based load balancing scheme. It uses a designated server, or dispatch manager, as a front-end to incoming requests. Clients send requests to a virtual IP address. The dispatch manager accepts incoming requests, parses the requested object id (URL in the case of Web requests), and re-directs the request to the appropriate server. The dispatch decision is based on object location and server load information. Cache servers, including the dispatch manager, exchange load information among themselves using a separate protocol. Once the request is forwarded to a server, that server responds directly to the client, bypassing the dispatcher.

2.2 Router-Based Load Balancing: Transparent Caching

An advantage of transparent caching over software-based load balancing mechanisms is that it cannot easily be bypassed. As the name implies, transparent caches partition web traffic transparently to the user. This means that proxy configuration is not needed, although both schemes can complement one another. Even if a sophisticated user disables proxy auto-configuration, a transparent cache captures “runaway” requests and re-directs them appropriately.

Transparent caching can be implemented by either: modifying the cache’s TCP stack so that it operates in promiscuous mode to capture all possible IP addresses; or, using the router to map a client’s TCP session to an appropriate Web cache. In the latter approach, the router acts as a switchboard between clients and the target Web caches: it captures a client’s TCP SYN packet, establishes a connection with the client, selects a Web cache, and establishes a separate connection with it. Upon establishing the two TCP sessions, the router forwards the client’s TCP packets to the selected cache by performing the appropriate mappings.

Several transparent caching schemes have been proposed; we review some of them below.

---

1 Although most of its customers use it as a load balancing scheme for Web servers, Dispatch can also be applied to Web cache servers.
Cache Director

Cisco’s cache director system [3] uses router support to intercept HTTP requests (TCP traffic destined to port 80) and partition them among a collection of cache engines. Each cache engine is assigned a share of the 256 address groups into which the IP address space is split. The router and cache engines run the Web Cache Control Protocol to monitor cache load and to re-allocate the IP address space accordingly. Cache engines keep statistics on address subgroup hit rate as a metric for cache load. The cache director uses these statistics to dynamically re-allocate high traffic address groups from heavily loaded to more lightly loaded caches. Address groups are also re-allocated when caches are added or removed (including cache failures) from the cache farm.

We are currently designing and implementing an alternative to the cache director scheme. The main idea behind our transparent caching scheme is to use an existing routing protocol (e.g., RIP) to advertise and process cache load information. Caches use routing updates to send current load information to routers. The advantage of this scheme is that it does not need a special-purpose protocol to implement transparent caching. Because it uses existing, standardized, publically available routing protocols, our scheme will be more easily portable.

Content-Based Redirection

A step forward from the cache director approach consists of using a more elaborate decision making process in the router. Instead of just deciding whether incoming traffic is Web related or not, the router performs content-based re-direction. Routers parse requests and compose a key using the source and destination IP addresses, TCP port number, and the object tag (the URL in case of Web requests). They use this key to lookup a locally-maintained content-location mapping database. The content-location database uses location information to re-direct requests. At the target site, load information can be used to direct the request to the “best” available server.

The content-based redirection approach can be used for content servers in general, including Web caches. ArrowPoint Communications [1] is currently developing a line of network appliances based on the content-based redirection concept.

2.3 Transparent Caching Drawbacks

The strength of transparent caching is also its main weakness: it violates the end-to-end argument by exposing network routing to higher protocol layers and trying to circumvent it. This results in decreased robustness: when routes flap, transparent cache clients may experience broken Web pages as their HTTP/TCP connections are no longer end-to-end.

3 Translucent Caching

Translucent caching is another way to perform router-supported load balancing among cache servers. Because they do not break the end-to-end nature of TCP connections, translucent caches are robust and do not suffer from problems that may result from route asymmetry. The basic idea behind translucent caching is to direct a request for an object toward caches along the way from the client to the object’s origin server. It takes advantage of the fact that the network routing fabric routes the request along the best path from the source (client) to the destination (server). Routers along the way intercept the request (in ICP [12] or other caching protocol) and echo back the address of a close-by cache. The previous-hop cache can then send the request to the indicated next-hop cache.

Figure 1 sketches how translucency works. A client requesting an object sends the request to the corresponding client cache (or proxy cache) CC. If CC does not have the object it forwards the request to the object’s home server. Router R1 along the way intercepts the
request and sends back a reply containing C1 as the next-hop cache. CC then sends the request to C1. Notice that this time around, CC’s request does not get intercepted by R1. We describe how we bypass router interception in Section 3.1 below.

To avoid the extra round-trip time (RTT), cache servers may cache next-hop information for future use. Like any cached object, cached next hops can be assigned a time-to-live (TTL) so that they are refreshed periodically. This allows translucent caches to adapt to changes in cache server load. Clearly, the tradeoff in setting these TTL values is between trying to avoid the extra RTT and keeping up with load and object location dynamics.

Pointing to the next-hop cache becomes a load balancing issue. Routers tunnel intercepted ICP requests to a process that keeps a database of nearby cache servers. This process can be co-located with the router. The next-hop cache database is equivalent to the content-location information kept by routers in the content-based transparent caching scheme. Similarly to transparent caching, translucent caches can periodically inform routers of their current load. They can also tell routers what IP address ranges they are willing to service. Currently, we manually configure next-hop cache information. As an extension to our translucent caching implementation and as part of our work in transparent caching, we plan to use an existing routing protocol to communicate cache content and load information to translucent routers.

3.1 Design Issues

The translucent caching approach requires that routers:

- Filter packets containing a request for an object coming from a cache. In our implementation, we use the Internet Cache Protocol (ICP). Throughout this section, we
use ICP to refer to how caches communicate among themselves.

- Perform a lookup for the next-hop cache information, and
- Send the next-hop cache information back to the previous cache using ICP.

The previous-hop cache sends the request to the next-hop cache. Alternatively, the router can forward the request to the next-hop cache and send next-hop information back to the previous-hop cache for future use.

Cache translucency may cause loops: a cache request may keep being intercepted by a router indefinitely. Possible approaches to avoid looping are:

- Define a special ICP message type that bypasses router interception. In other words, if a cache already has next-hop cache information (received as the result of the current or a previous request), it will use the appropriate ICP message type to prevent the next-hop router from intercepting the request. This solution requires that routers process every cache request. In case it receives the bypass message, the router forwards the packet to its original destination.

- Another option is to use a different port number when bypassing router interception. This will automatically bypass the router filter without routers having to process every ICP packet.

For performance, we plan to restrict the number of next-hop cache lookups along the path between a source-destination pair. If a request goes through say 2 or 3 next-hop lookups without hitting a cache, it is forwarded straight to its destination, bypassing any other next-hop lookups. Take the scenario shown in Figure 1 with a maximum number of intercepts of 2. The client cache CC gets a request from a client. If it doesn’t have the requested object, it sends an ICP request towards the origin server. The first router along the way, R1, tells CC to try C1. CC sends the request to C1 using the “no intercept” message to bypass R1. If it is a cache miss, C1 increments number of intercepts to one. It checks whether the number of intercepts is less than the pre-specified maximum number of intercepts, in which case it forwards the ICP request towards the server. The request gets intercepted by R3 who tells C1 to try C4. C1 then sends the request to C4 using the ICP “no intercept” message. In case C4 does not have the object, it will increment number of intercepts to two. This causes C4 to send an HTTP request for the object to the origin server, instead of sending back next-hop cache information.

3.2 Implementation

We currently have a working prototype implementation of translucent caching under Solaris 2.5.1. Our code is available from http://www-scf.usc.edu/yousef/cache.html.

Our implementation of translucent caching works as follows. Routers filter ICP request packets using the IP packet filter package [8]. The filter operates for both inbound and outbound sides of the IP packet queue and checks packets before they get checked for source route options. The packet can be configured with a list of rules defined in the filter’s configuration file. These rules allow packet filtering by IP protocol, IP options, network interface, and port number. For example, the rule "block in log proto udp from any to any port = 3130 causes the filter to intercept all incoming UDP packets coming from any host and destined any host at port = 3130. The intercepted packets are redirected to /dev/ipf.

We chose to define new ICP message types instead of assigning a new Squid port number. We modified the ICP protocol and incorporated two previously undefined ICP messages. To the first one, we assigned opcode 12 and use it to convey next-hop cache information
ICP reply with next-hop cache address payload format (OPCODE = 12):

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Version</th>
<th>Message Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Request Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Option Data (containing Hop Count)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sender Host Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Next-hop Cache Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Null-Terminated URL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ICP request with no-interception payload format (OPCODE = 13):

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Version</th>
<th>Message Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Request Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Option Data (containing Hop Count)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sender Host Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requester Host Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Null-Terminated URL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: New ICP messages.

to the previous hop. The second new ICP message was assigned opcode 13 and is used to bypass translucent routers. Figure 2 shows the new messages' format. Note that the new messages' option data header field contain number of intercepts. The option data field in ICP’s opcode 1 message (ICP_OP_QUERY) header was also modified to contain number of intercepts information.

We incorporated our translucency protocol into the publicly available Squid Internet Object Cache [11] (Squid version 1.1.14). We modified the Squid server to perform the additional functions listed below. Figure 3 shows the Squid pseudocode including our modifications.

- Generate an ICP reply to send next-hop cache information to the previous hop cache. We implemented this new message using ICP’s previously undefined opcode 12 message.

- Recognize the new ICP opcode 12 message and use the next hop cache information it contains to generate the next ICP request for the object, in this case the ICP opcode 13 message.

- Generate the new ICP opcode 13 message and send it to the next-hop cache. When a translucent router intercepts an ICP opcode 13 message, it regenerates the request with the same destination address.

- Recognize the new ICP opcode 13 message and generate a regular, ICP opcode 1 message if the object is not locally cached. This request will be intercepted by the next-hop router.
(1) If an HTTP, Gopher, etc. request from client
   {
     If object found in local cache
     {
       Transfer the object to client;
     }
     else
     {
       Send ICP request with OPCODE = 1 towards the source of object;
     }
   }

(2) If ICP packet with OPCODE = 12 /* containing next-hop cache */
   {
     Send ICP request, with OPCODE = 13 for no interception, to next-hop
     cache address found in message payload;
   }

(3) If ICP request packet with OPCODE = 1 or OPCODE = 13 /* normal or no
     interception request */
     {  
       if object found in local cache
       {
         reply to previous cache with ICP_HIT;
       }
       else
       {
         number of intercepts++;
         If (number of intercepts == maximum number of intercepts)
         {
           Get object directly from source and transfer it to client proxy
           cache;
         }
         else
         {
           Send ICP message ICP_MISS to previous cache;
           Send ICP request with OPCODE = 1 towards the source;
         }
     }
   }

(4) If ICP reply packet with ICP_HIT
   {
     Get object from the cache sending ICP_HIT and transfer it to client
     proxy cache;
   }

Figure 3: Modified Squid pseudocode.
for(;;)
{
    Read the intercepted packet

    If ICP request packet with OPCODE = 1
    {
        Get the next-hop cache address from cache directory;
        Send back an ICP message with OPCODE = 12, containing next-hop cache
        address, to the host address in 'Sender Host Address' field of ICP
        header (i.e., previous-hop cache);
    }

    If ICP request packet with OPCODE = 13
        Regenerate the request towards the original destination;
    }

    Figure 4: ipmon pseudocode.

    • Process number of intercepts. If incoming request’s number_of_intercepts <
      maximum_number_of_intercepts, increment request’s number_of_intercepts, and for-
      ward ICP opcode 1 (ICP_OP_QUERY) request. Otherwise, generate HTTP request to the
      object’s origin server. The current implementation uses maximum_number_of_intercepts =
      3.

    The modifications made to the router are listed below.

    • Install and run the IP packet filter (we use IP packet filter version 3.1.11 in our current
      implementation) in the kernel to intercept ICP messages, i.e., messages destined to
      port 3130.

    • Run a user-level background process, ipmon. The ipmon process, whose pseudocode
      is shown in Figure 4, reads intercepted packets from /dev/ipf and processes them as
      follows:

        – If the ICP message opcode is 1, then generate an ICP opcode 12 message with
          the next-hop cache information to the previous-hop cache.

        – ipmon looks up the next-hop cache host name from a cache database maintained
          in the router. In the current implementation, this next-hop cache database is
          manually configured and lookups use round-robin to select the appropriate entry.

        – If ICP message opcode is 13, then forward the message towards its original des-
          tination.

    We use the scenario in Figure 5 to demonstrate how our prototype implementation works.
    The client running on excalibur.usc.edu sends an HTTP request to its proxy server also
    running on excalibur.usc.edu. The proxy server does not have the requested object so it
    generates an ICP request for the object. This ICP request is intercepted by the translucent
    router dorado.usc.edu. The router looks up the address of the next-hop cache (in this
    example, paloma.usc.edu) and sends it back to the previous-hop cache excalibur.usc.edu
    using ICP’s opcode 12 message. When the cache gets an ICP opcode 12 message, it knows
    that the payload contains the address of next-hop cache. excalibur.usc.edu then sends an
    ICP opcode 13 request to paloma.usc.edu with number_of_intercepts = 1. Since paloma
does not have the object it generates an ICP request towards the object’s origin source. This request is intercepted by another translucent router cabrillo.usc.edu, who returns the address of the next-hop cache (here, jalama.usc.edu). paloma.usc.edu then sends and ICP opcode 13 request to jalama.usc.edu with number of intercepts = 2. Since the object is not cached locally, jalama increments number of intercepts and decides to fetch the object directly from the server.

3.3 Addressing Availability and Fault Tolerance

The fact that users and services rely heavily on caching for proper Internet connectivity also means that cache failures can be catastrophic, and must be avoided.

Several of the load balancing schemes discussed in Section 2 also act as fault tolerance solutions. In proxy auto-configuration for example, the configuration program can specify a primary and a secondary cache server. In case the primary cache is unavailable, the browser automatically forwards the request to the secondary cache.

We address fault tolerance using a clustering-based solution. The idea is to organize collaborating translucent caches in clusters of two or more cache servers. At any given time, one of the cluster members is operating as the cluster representative, who is responsible for answering requests received by the cluster. The remaining members of the cluster operate as backups. If for any reason the cluster’s representative becomes unavailable, a backup will automatically take on the representative role and will start answering requests on behalf of the cluster.

Clients reference a cache cluster using the cluster’s IP address, an IP address to which

---

A similar solution has been proposed for achieving fault tolerance in routers [6].
the virtual interfaces of all cluster members are bound. Cache cluster clients such as web browsers will be configured to point to a cluster the same way as they point to regular caches.

3.3.1 The Cluster Failover Protocol (CFP)

For simplicity, we assume a cluster of two caches. The resulting protocol can be easily extended to handle larger clusters, which is an item for future work. The cluster pair runs the cluster failover protocol (CFP) which allows the backup cache to detect failure of the cluster representative and to take on its role.

At startup, one of the caches in a cluster is configured as the cluster representative. Alternatively, the cluster can go through an initialization phase, where the cluster caches start up as backups. Depending on their configured priority, one of the caches assumes the role of the cluster representative.

The backup cache periodically polls the representative by sending standard ICMP request messages (or ping's) and waiting for replies. If for some reason the representative does not respond, the backup assumes it is down and becomes the cluster representative. Once the default representative comes back up, it resumes its functions and the backup cache goes back to its default backup state.

At any given time, there is only one cluster representative who responds to requests destined to the cluster. Requests are addressed to a virtual IP address, or the cluster’s IP address, which is bound to the virtual interface of both cluster caches. The representative has that interface turned on, while the backup cache has it turned off. When the backup cache takes on the representative role, it turns on the virtual interface bound to the cluster's IP address. It also sends a gratuitous ARP broadcast to directly connected routers and gateways. The ARP broadcast causes the appropriate routing table MAC-IP address mappings to be updated. This way future client requests will be delivered to the current representative.

When the default representative comes back up, it sends an ARP broadcast to update directly connected routers. The representative also sends periodic ARP broadcasts to ensure ARP records are up-to-date.

3.3.2 Implementation

We implemented the representative and backup side of the protocol using a single perl script. Our code is available from http://www-scf.usc.edu/~arthachi/clusterd-doc/). Through a command line parameter, the cluster administrator configures one of the cluster members as representative or as backup. Figure 6 shows the protocol’s pseudocode.

Besides configuring a cache as representative or backup, the cluster administrator can also use command line parameters to configure the parameters below. Otherwise, they take on their default values.

- Path names for perl, ping, and ifconfig. Their default values are /local/bin/perl, /usr/sbin/ping, and /sbin/ifconfig.
- Name of virtual interface bound to cluster IP address.
- Cluster IP address.
- Cache server IP address.
- Ping timeout period, with default value set to 3 seconds.
- Poll interval with default value set to 3 seconds.
for(;;)
{
    /* Backup cache */
    if cache is BACKUP
    {
        if (ping is successful)
        /* Representative cache is up. */
        {
            /* Switch from representative to backup or stay as backup. */
            ifconfig virtual interface DOWN
        }
        elseif (ping timed out)
        /* Representative cache is down. */
        {
            /* Switch from backup to representative. */
            ifconfig virtual interface UP
        }
        else (ping failed)
        exit
    }
    /* Representative cache */
    else
    {
        /* Send gratuitous ARP broadcast. */
        ifconfig virtual interface UP
    }
    sleep POLL_INETRVAL;
}
The ping timeout interval is the interval in seconds the backup waits for an answer from the cluster representative before declaring it to be down. We set the current default value to 3 seconds. The frequency at which the backup cache polls the representative is given by 1/poll interval. The poll interval's default value is currently set at 3 seconds. The cluster administrator can configure the timeout and poll interval depending on how responsive to cache failures the cluster should be.

4 Conclusions

In this paper, we propose translucent caching as an alternative to transparent caching-based load balancing. Translucent caches are robust: by not ‘‘splitting’’ the TCP connection between the client and a cache, it avoids the problems that may be caused by route flapping and asymmetry.

Translucent caching uses routers along the best path between a client and a server to direct requests toward nearby, lightly loaded caches. We presented our translucent caching design and implementation. We also presented a clustering-based failover implementation that can be used by translucent caches for improved fault-tolerance and availability.

References


\footnote{Note that our default value overwrites ping's default value of 20 seconds.}