

Yousef Balout & Abdullah Dawood

Adviser: Dr. Othman Othman

2016 – Fall Semester – Computer Science

Sentro

Transparent Cache Server

Contents

[Abstract 2](#_Toc449976895)

[Introduction 2](#_Toc449976896)

[Competitive Solutions 2](#_Toc449976897)

[Design goals 4](#_Toc449976898)

[Design 5](#_Toc449976899)

[Arp Spoofer 5](#_Toc449976900)

[Traffic Controller 6](#_Toc449976901)

[Connection Controller 6](#_Toc449976902)

[Cache Store 8](#_Toc449976903)

[Metadata Database 10](#_Toc449976904)

[Cache Management 10](#_Toc449976905)

[General Notes 12](#_Toc449976906)

[Limitations 12](#_Toc449976907)

[Future Work 12](#_Toc449976908)

[Conclusion 12](#_Toc449976909)

Sentro

Transparent Cache Server

# Abstract

This document describes a proof of concept of a transparent cache server that is intended to improve the performance of small-medium networks for people who have a no or minimum networking knowledge.

The project aims to provide caching solution for web content (images, sound, videos, and static files) to reduce bandwidth usage as well as increasing the download speed for cached content.

The project target users with no network experience so we introduce a use of ARP Spoofing hacking as a solution to provide a higher layer to hide the proxy and reference it as Transparent.

The design is not an optimal solution but a proof of concept of using hacking attacks as a solution, also it provide an easy, low cost solution.

The document covers the part which are done during the semester phase and doesn’t go in details with non-implemented parts.

# Introduction

Caching is the technique of keeping frequently accessed information in a location close to the requester. A Web cache stores Web pages and content on a storage device that is physically or logically closer to the user and faster than a Web lookup.

The project aims to provide a solution to the monthly bandwidth cab set by ISPs, most users exceeds this cab and have to pay extra fee to continue using the internet, we are trying to reduce the usage of bandwidth by storing web content in a local computer and deliver them to users on request from the local network instead of internet and thus reducing bandwidth usage.

### Competitive Solutions

The problem itself is not new and there are plenty solutions available out there each of which targets a specific users or use cases, see the next table to know where this project stands in the competition.

|  |  |  |
| --- | --- | --- |
| Solution | Pros | Cons |
| Caching Hardware | * Highest Performance * Scalable * Reliable | * High Cost * Hard Deployment * Require High Network knowledge * Lost Internet Connection on device shutdown |
| Caching Proxy | * Low Cost * High Performance * Large Storage * Scalable * Reliable | * Hard To Configure * Long Deployment Time * High learning curve * Lost Internet Connection on software shutdown * Override Router Traffic Engineering Features |
| Transparent Caching Server | * Low Cost * Fast Deployment Time * Internet Connection Not Lost on software shutdown * Extend Router Traffic Engineering Features * Large Storage * Easily Extended Storage * Require No Network Knowledge | * Not Scalable * Not Reliable * Average Performance |

Hardware cache servers are just a caching proxies installed on a custom hardware designed just for caching and traffic forwarding to outcome with the highest performance. Some of famous vendors are:

|  |  |
| --- | --- |
| Name | Vendor |
| Cache Engine | Cisco |
| CacheFlow Series | CacheFlow |
| CacheRaq 2 | Cobalt Networks |
| NetCache | Network Appliance |

Software proxy servers are the most famous and wide spread, the cons “**Long Deployment Time**” mentioned above refer to the process where every device in the network must be configured to use the specified proxy server which could be very time consuming with tens or hundreds of devices connected, and because every device is configured to use the proxy there will be no internet connection once that proxy is shut down.

A set of the most known proxies are:

|  |  |  |
| --- | --- | --- |
| Name | Vendor | Platform |
| Apache Web Server Caching Module | Apache Information Services | AIX, BSD/OS, Digital UNIX, FreeBSD, HP-UX, IRIX, Linux, NetBSD, NextStep, SunOS, Solaris, SCO UNIX, Windows NT |
| DeleGate | MITI ETL | AIX, EWS4800, HP-UX, HI-UX, IRIX, NextStep, NEWS-OS, Digital UNIX, Solaris, SunOS, BSD/OS, FreeBSD, Linux, NetBSD, OpenBSD, Windows 95/NT, OS/2 |
| Squid | NLANR | AIX, Digital UNIX, FreeBSD, HP-UX, IRIX, Linux, NetBSD, NextStep, SunOS, Solaris, SCO UNIX, OS/2 |
| Traffic Server | Inktomi | Digital UNIX, FreeBSD, IRIX, Solaris, Windows NT |

As said, proxy server are hard to use and configure, our solution work around these by using ARP Spoofing hack technique which exploits a security issue in Address Resolution Protocol which maps IP address to MAC addresses. ARP Spoofing is used to automatically redirect traffic from all machines in the network to the cache proxy server that provide/cache web content, thus Transparent to users.

Another interesting point is the work the proxy itself work, instead of traditionally creating a new connection to the destination address we use raw socket programming to simulate the connection and make the end user appear as it is connected directly to the destination address, thus Transparent to the router.

Being Transparent to the router has its benefits, it doesn’t override router traffic polices applied to network users. As an example, the router has a policy to block a website for user A, in a normal proxy the router will see all the connection coming from the proxy server and can’t distinguish user A. In our solution, we simulate the connection and modify packets to traffic from user A will always be seen as from user A, thus the router polices are still applied.

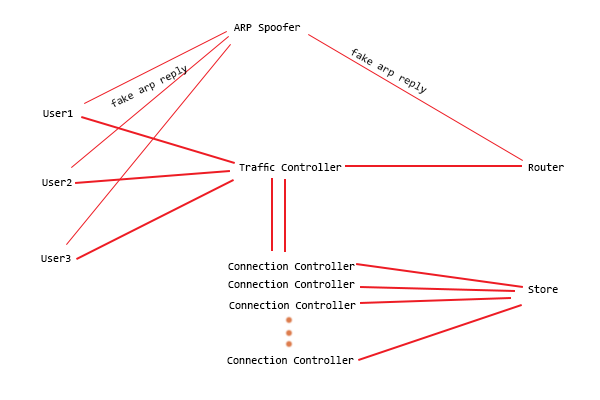
# Design goals

Main points considered during the solution design are:

* Target home users who have no network knowledge.
* Target small networks.
* Users should not be aware of the presence of the application.
* Router should not be aware of the presence of the application.
* Minimum or zero configurations needed to get started.

# Design

The main idea is to redirect traffic to cache server without user interaction, so ARP Spoofing attack is used, then the cache server will handle all the traffic as like a proxy server.

 The big picture of the process

### Arp Spoofer

The ARP Spoofing is a traditional attack involve sending ARP REPLY messages with fake information to targeted users, targeted devices will update their ARP Table and send their traffic to cache server instead of router which must be attacked as well.

Steps:

1. Send spoofed ARP REQUEST message to router and target.
2. Send spoofed ARP REPLY message to router and target.
3. Wait 10 – 30 seconds.
4. Go to 2.
5. If stopped -> Send real ARP REQUEST message to router and target.
6. Delete all entries in ARP Table of the hosting machine.

Once the attack is started you should spawn an ARP detector thread that watch the network for ARP messages and analyze them to detect any message that could defect the attack, and ARP Spoofer must take action immediately once this message is detected to keep the target spoofed all the time.

Notes:

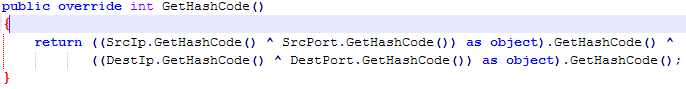
* Arp attacker must change the target devices mac addresses from dynamic mapping to static mapping.
* Arp attacker must disable ICMPRedirect through registry.
* Arp attacker must enable IPEnableRouter through registry.

### Traffic Controller

One main issue the traffic controller solve is redirecting traffic to the application layer before it is forwarded to the router, other solutions like WinPcap or similar sniffers doesn’t work because they sniff a copy of the packet after it been forwarded, so we used a feature introduced first in Windows Vista which called “Windows Filtering Platform (WFP)”, and thus the application doesn’t work on any Windows OS without this feature.

Afterward, the traffic controller reads the IP and Port addresses from the packet and pass it to the right Connection Controller where the state of each connection is saved.

To uniquely identify each connection we could use an XOR of all the addresses



Traffic controller is supposed to be easily extended to support more features other than redirect packet to the right connection controller, such as blocking specific user from accessing internet at some schedule for example.

### Connection Controller

The connection controller holds most of the logic of the application since it simulate the TCP connection between client and destination server

Because we use WPF in the application we have to follow its rules, one important one is Calculating Checksums. In ordinary application, the checksums are set 0 and then it is calculated by network card, but in our case WPF refuse send any packet without checksums, so every packet must have a valid checksum before it is sent in the network.

With high speed of local network we have we can send files very quickly which is good, and bad. The fact that there is a limit of how much could a user’s CPU process, if we sent content very fast then the user will not be able to process all of it, and will most likely drop some packets. So the application should be always aware of the window size updates in ACK packets sent by user and never send packets more than user capability.

There is a case in http 1.1 protocol when the keep-alive header is set and the connection is kept open after the end of transmission, which mean client send multiple request in the same connection. When a request is cached and sent there will be a different ACK and SEQ for server and client which implies that they can’t continue talking in this connection since they are out of sync, hence there is must be a synchronization between them.

To keep SEQ and ACK numbers synchronized the connection controller must add the number of bytes sent from cache to the ACK number of packets passed to the server, and add the number of bytes consumed by connection controller to SEQ number of packets outgoing to server. The same thing apply to packets incoming from server and outgoes to the client.

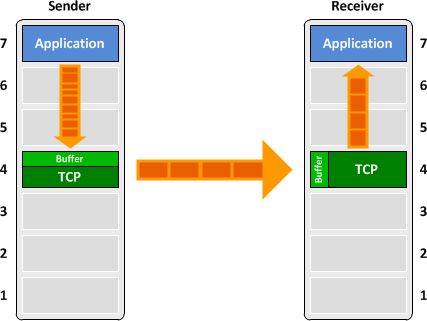
However, synchronizing doesn’t completely solve the issue, when multiple request have been sent in the same connection and all of them were cache hit that mean the server has opened a connection and didn’t receive any request! Thus the server will send FIN packet and terminate the connection from its end, which means the client can no longer send any request to the server if request was cache miss, so to keep the server connection open it will be the connection controller responsibility to send keep-alive TCP packet often. Keep-alive packet is a packet with single byte in TCP data and a SEQ number less than the current SEQ number by one.

The application is very IO bound with very little CPU processing, this arise the vitality of implementing multi-threaded processing. But multithreading is not sophisticated for IO bound activities, if we say there is a thread for every packet which may implies IO or may not then we will run out of threads in a short time, anyway thread pool is a second and better solution, but also is not efficient since the number of packets is enormous. The best way to handle this is by using Asynchronous Programming which implement a state machine behind the scene. Asynchronous allow the application to make multi-threaded behavior in single thread! When an IO is requested then the thread doesn’t wait for IO to finish, it will break the state and do another processing, and when the IO is finished it continues from where it has stopped.

After all, if you notice abnormal performance behavior where the cached content is delivered but not consumed by the browser immediately then you must remember to set PSH flag in the first and last packet to avoid this performance penalty.

To understand the function of the PSH flag, we first need to understand how TCP buffers data. TCP operates at layer four of the OSI model; it presents to upper layers a simple socket which can be read from and written to, masking the complexities of packet-based communications. To allow applications to read from and write to this socket at any time, buffers are implemented on both sides of a TCP connection in both directions.

The diagram below shows how data is buffered by the sender before sending, and by the receiver upon reception.



Buffers allow for more efficient transfer of data when sending more than one maximum segment size (MSS) worth of data (for example, transferring a large file). However, large buffers do more harm than good when dealing with real-time applications which require that data be transmitted as quickly as possible. Consider what would happen to a Telnet session, for instance, if TCP waited until there was enough data to fill a packet before it would send one: You would have to type over a thousand characters before the first packet would make it to the remote device. Not very useful.

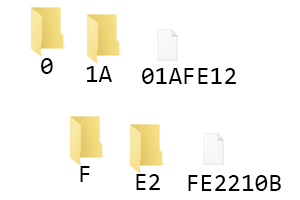
This is where the PSH flag comes in. The socket that TCP makes available at the session level can be written to by the application with the option of "pushing" data out immediately, rather than waiting for additional data to enter the buffer. When this happens, the PSH flag in the outgoing TCP packet is set to 1 (on). Upon receiving a packet with the PSH flag set, the other side of the connection knows to immediately forward the segment up to the application. To summarize, TCP's push capability accomplishes two things:

* The sending application informs TCP that data should be sent immediately.
* The PSH flag in the TCP header informs the receiving host that the data should be pushed up to the receiving application immediately.

One last gotcha the connection controller must take care of is the Ethernet padding, where Ethernet (802.3) requires every packet to be 60 bytes length at least, which means the connection controller must set 0s at the end of Ethernet layer until the packet size is 60. But since we are using WPF we can’t modify the Ethernet layer, as a walk around solution add the padding at the start of IP layer, which will work too but make sure they are added after checksum calculation.

By the end, don’t forget that the network is Big Endian and Windows OS is Little Endian, so reverse byte order is required before sending in network.

### Cache Store

The cache store was designed to be easily deployed, fast lookup and simply managed, among of many choices we decided to make a file store where each file name is a unique hash of the content URL, the file then is stored in a folder hierarchy that works as an index. The first tier in the hierarchy consist of 16 folders named from 0 to F, the second tier consist of 255 folders named from 00 to FF, the last tier is where files stored with names consist of 8 digits hexadecimal ranges from 0 00 00000 to F FF FFFFF. This implies that the store can hold up to 4 billion URLs.

The design is similar to the browsers cache store, except it works better for large size files (browsers usually limit maximum cached file size to 10MB).

The hashing function used was meant to be fast, low collision rate and not designed for security purpose, these criteria’s lead to “Murmur2” hash function witch result hash consist of 8 Hex values, we tested it on a 50 thousands URLs and there were no single collision.

Before hashing the URL to be looked up or stored in the cache store there is a must step to avoid inconsistent hashing output, we need a URL normalization. This process is solve the issue where a multiple URLs references the same content, for example:

<http://www.example.com/display?lang=en&article=fred>

<http://www.example.com/display?article=fred&lang=en>

http://example.com/display?article=fred&lang=en

<http://example.com/display?article=fred&lang=en#section1>

All of these are mapped to the same content but result of different hash codes, so normalization process aim to rewrite the URL is a standard form.

Some of the normalization steps are:

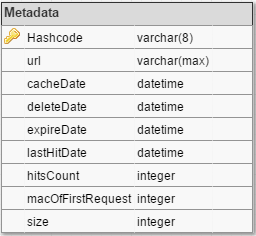
Removing fragments, removing duplicate slashes, adding “www” as the first domain label, sorting the query parameters, removing the "?" when the query is empty.

The design has its Pros and Cons, for example: the indexing is logical and doesn’t allocate any additional space to store indices (which may grow very large), another Pro is it is simple to implement, manage and moving from storage to another. But on the other hand, the large number of files make the deletion process very slow, also there is a lot of wasted space due to OS block size and clustering, which means a defragmentation should be run often.

The cached file contains the complete response from the origin server, included the response header and data, plus extra 2 bytes at the beginning of the cached file. These 2 bytes indicate the length of the response header which is read first and sent alone in a separate packet with PSH flag is set, this is important to allow the client browser to recognize the size of the income response and how large is it. This is also important to avoid performance penalty described above.

### Metadata Database

To keep track of all the cached files, or to execute some analysis we could traverse all the store easily, but that is a pure madness, so we keep a metadata(summery) of all the cached files in a simple database (SQLite) with single table, the design of the database is:



All the dates are indexed plus the macOfFirstRequest, this simple table allow to track all the cache store and help deciding which files are to delete to free space

### Cache Management

The caching rules is almost straight forward by following the rules of **RFC 7234 (Hypertext Transfer Protocol (HTTP/1.1): Caching)** this RFC describe the whole process of caching the web content, cautions and best practices during implementation.

Through various iterations of the HTTP protocol, a few different cache-focused headers have arisen with varying levels of sophistication. The ones we need to pay attention to are below:

* **Expires**: The Expires header is very straight-forward, although fairly limited in scope. Basically, it sets a time in the future when the content will expire. At this point, any requests for the same content will have to go back to the origin server. This header is probably best used only as a fall back.
* **Cache-Control**: This is the more modern replacement for the Expires header. It is well supported and implements a much more flexible design. Some of the Cache-Control options you can use to dictate your content's caching policy are:

1. **no-cache**: This instruction specifies that any cached content must be re-validated on each request before being served to a client. This, in effect, marks the content as stale immediately, but allows it to use revalidation techniques to avoid re-downloading the entire item again.
2. **no-store**: This instruction indicates that the content cannot be cached in any way. This is appropriate to set if the response represents sensitive data.
3. **public**: This marks the content as public, which means that it can be cached by the browser and any intermediate caches. For requests that utilized HTTP authentication, responses are marked private by default. This header overrides that setting.
4. **private**: This marks the content as private. Private content may be stored by the user's browser, but must not be cached by any intermediate parties. This is often used for user-specific data.
5. **max-age**: This setting configures the maximum age that the content may be cached before it must revalidate or re-download the content from the origin server. In essence, this replaces the Expires header for modern browsing and is the basis for determining a piece of content's freshness. This option takes its value in seconds with a maximum valid freshness time of one year (31536000 seconds).
6. **s-maxage**: This is very similar to the max-age setting, in that it indicates the amount of time that the content can be cached. The difference is that this option is applied only to intermediary caches. Combining this with the above allows for more flexible policy construction.
7. **must-revalidate**: This indicates that the freshness information indicated by max-age, s-maxage or the Expires header must be obeyed strictly. Stale content cannot be served under any circumstance. This prevents cached content from being used in case of network interruptions and similar scenarios.
8. **proxy-revalidate**: This operates the same as the above setting, but only applies to intermediary proxies. In this case, the user's browser can potentially be used to serve stale content in the event of a network interruption, but intermediate caches cannot be used for this purpose.
9. **no-transform**: This option tells caches that they are not allowed to modify the received content for performance reasons under any circumstances. This means, for instance, that the ca–che is not able to send compressed versions of content it did not receive from the origin server compressed and is not allowed.

* **Etag**: The Etag header is used with cache validation. The origin can provide a unique Etag for an item when it initially serves the content. When a cache needs to validate the content it has on-hand upon expiration, it can send back the Etag it has for the content. The origin will either tell the cache that the content is the same, or send the updated content (with the new Etag).
* **Last-Modified**: This header specifies the last time that the item was modified. This may be used as part of the validation strategy to ensure fresh content.

### General Notes

In the asynchronous programming model suggested with C# language we must insure order of packets sent in the network or written to the store, and hence we need a synchronization method, the best tool available is SlimSemaphores afforded by “.Net” framework, SlimSemaphores has a nano seconds impact on performance which is negligible compared to other methods or tools for the same purpose.

Also in order to insure single process of the application at time we could use a Mutex to make synchronization across processes.

Lastly, we provide two interfaces to interact with cache server, first is a command line and second is a Web UI with elegant modern design. To make a support for the Web UI we host a web server in the application process using OWIN/Katana self-hosting technology.

# Limitations

In favor of simplicity we have another drawbacks and limits that come up with the solution

#### Scalability

The design of forwarding all the traffic to single machine that is not designed for heavy traffic forwarding/processing make the performance degrade with each active device connected to the network and thus not scalable.

#### Security

In case of any website say “website.com” is blocked via router for a specific user then there is a chance that site is opened and main page is cached and thus the website is delivered to the blocked user and bypass the router blocking policy.

# Future Work

The main purpose of the project is caching, but it can be extended to provide more features

* SSL Caching: by on-fly certificate generation with installed root certificate.
* Monitoring: track every connection and users activities.
* Controlling: allow/block websites and limit users’ internet access.
* Browsing Cache: allow interacting with cached content via browser.

# Conclusion

Sentro, Transparent Cache Server tries to provide a new solution to network latency and bandwidth limits by combining a caching solution with a hacking technique to provide simplicity and efficiency to small networks of users who share common interests such as families.

The project provide a transparent solution to maximize simplicity and keep users away from messing with technical stuff. However, performance is traded off with simplicity a little bit.