Flying Squid

Tufts University: Senior Design Project

Advisor: Prof. Fahad Dogar

Sunjay Bhatia, Victor Chao, Jim Mao, and Siddhartha Prasad

Spring 2016

Flying Squid is a proxy that can provide a distributed, cloud based cache localized at the edge of a network. The system can eliminate unnecessary bandwidth use, and provide faster data delivery than content providers by themselves. Flying Squid aims to be a truly web-based personal proxy, with shared cloud based caches and value based partial caching. By improving content delivery speed and reducing client data usage, the proxy will be especially useful to mobile users and those with spotty network connectivity.

Contents

1 Background

Proxy servers help communication between two entities on a network. They often act as intermediaries between clients and content-delivering servers. They help optimize and add structure to networks and distributed systems. The latency incurred from downloading content directly from target servers is fast becoming a limiting factor on internet speeds. With sharp upward trends in the number of devices connected to the internet, this has forced cell-phone providers, ISPs and even large local area networks to use these servers to optimize both networks and content delivery. As a result, end users consciously and unconsciously end up using several proxies a day. While these proxies have sophisticated caching and compression mechanisms, they are stand-alone programs that do not do the most they can to share information about users and the data they are caching.

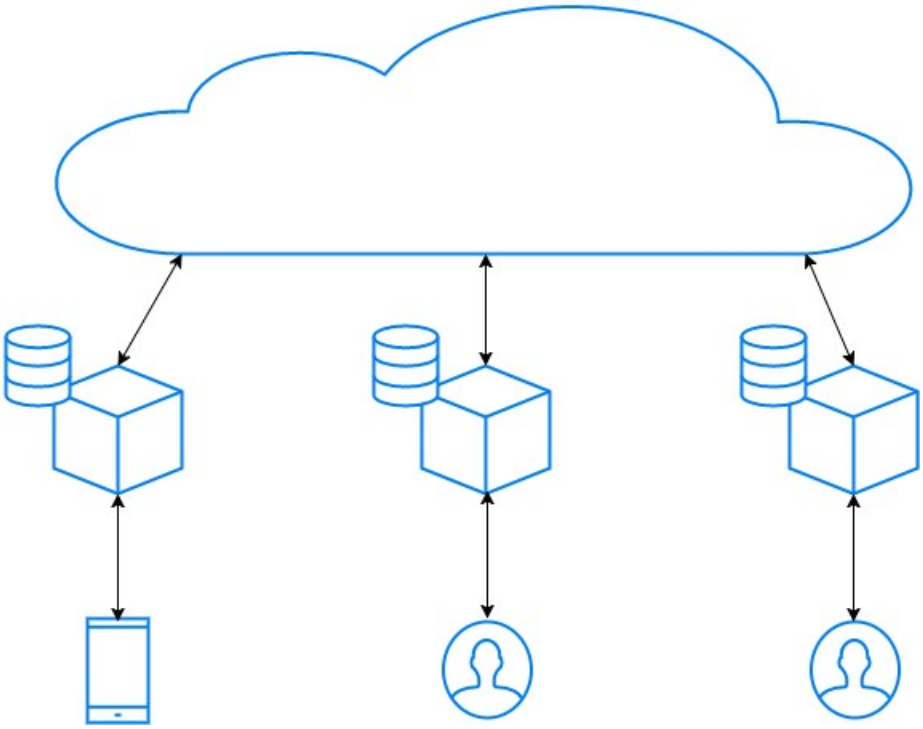

Figure 1: Traditional Client-Proxy Architecture

In Figure 1 we can see a simple representation of the traditional role proxies have. Users connect to proxy servers as intermediaries to their connection to a network, such as the internet. Each of these proxies traditionally has its own disk and RAM cache. It is also possible to utilize these proxies as a distributed

network with a shared cache represented by ‘joining ’each of the disk caches into a unified cache. However, this cache still only incorporates the basic levels of disk and RAM storage and there is potentially significant overhead in communicating between proxies to locate content that may be on a different physical machine than the one handling the request and serving the content.

2 Flying Squid

2.1 Apache Traffic Server

Apache Traffic Server (ATS) is a high-performance open-source proxy that was built by Inktomi and Yahoo!. It is modular in nature, and is known for its efficient caching mechanisms. According to the ATS documentation, it “is designed to improve content delivery for enterprises, Internet service providers (ISPs), backbone providers, and large intranets by maximizing existing and available bandwidth ”[1].

Flying Squid is an Apache Traffic Server augmentation that can provide a distributed, cloud based cache localized at the edge of the network. The system utilizes custom caching techniques and protocols to reduce uneccesary bandwidth use. Thus, it provides faster data delivery than content providers by themselves.

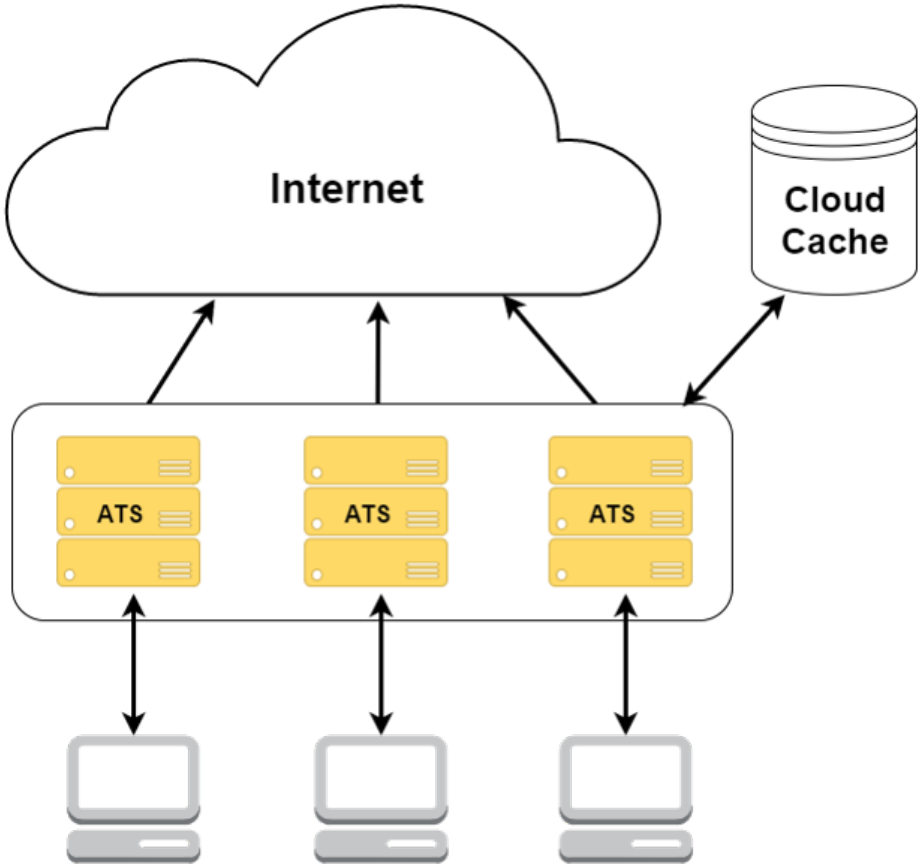

Figure 2: Flying Squid Architecture

Flying Squid aims to be a truly web-based personal proxy, with shared cloud based caches and value-based storage and transfer protocols. By improving content delivery speed and reducing client data usage, the proxy is especially useful to mobile users and those with spotty network connectivity.

3 Cloud Caching

FlyingSquid leverages different cloud storage tiers, each with unique benefits, to build a more sophisticated caching infrastructure. As a result, individual instances in an ATS cluster can be configured to share access to cached objects and use storage space more efficiently.

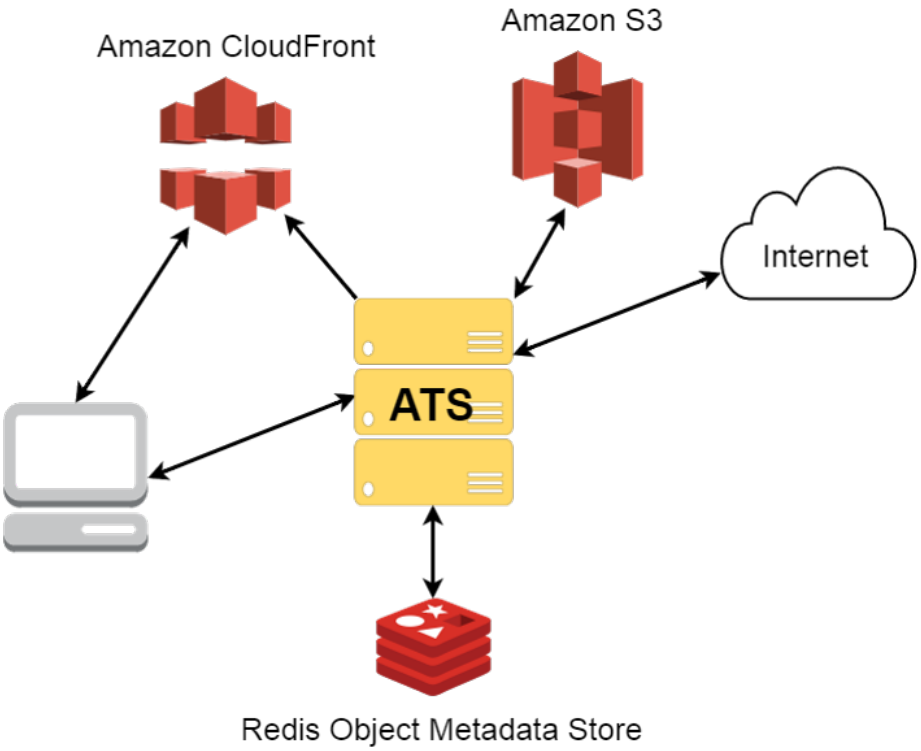

Figure 3: Flying Squid Proxy Cloud Caching Architecture

3.1 Augmented Tier - Redis Cluster

FlyingSquid uses a Redis in-memory key/value store to save HTTP object metadata. Each member of an ATS cluster has a master and slave pair of Redis instances. These instances are connected as a cluster and objects are stored across these instances according to how the cluster partitions keys. We chose to use this storage format for object metadata as opposed to storing data with each object because we needed a faster method of accessing data than querying S3 (where all cache objects are ultimately stored), as this type of query will inherently introduce significant performance penalties.

FlyingSquid uses the ATS computed cache key hash for each object as the key to store metatadata under and serialize the struct below to a string to store it in Redis.

```
struct ObjectCacheMeta {
    int64_t size; // Object size in bytes
    int64_t headerLength;
    char *responseHeader; // Response header to send back to client
    long lastAccessed; // Unix timestamp of when object was last accessed
    long cloudFrontExpiry; // Unix timestamp of when object will be
        removed from CloudFront
  };
```

Whenever we do a cache write we insert or update this object metadata. When we do cache reads and access objects in the cache, we update the lastAccessed timestamp and if applicable update the cloudFrontExpiry field.

3.2 Storage Tier - AWS Simple Storage Service (S3)

FlyingSquid utilizes Amazon S3 public cloud storage as the largest and slowest tier of the cloud cache. An ATS cluster is configured to share storage of HTTP objects in a S3 bucket. When ATS determines that an HTTP object is cacheable the object is uploaded to S3. When a cache read is called for we determine whether an object is in S3 from the Redis object metadata. If so, we simply retrieve it, collect the header from Redis, and send it back to the client.

3.3 Storage Tier - AWS Cloudfront

CloudFront is Amazon's Content Delivery Network service which can be utilized to push content to Amazon's servers at the edge of the network. FlyingSquid uses it to speed up object delivery to the client. When an object in the cache is deemed to be popular we set a field in its S3 metadata to push it to CloudFront. This field is equivalent to the HTTP 'Expires 'header and determines the lifetime of the object in this highest cache tier. When an object in CloudFront is requested by the client, an HTTP redirect is used to serve the response.

4 Fingerprinting: Value Based Caching

Whether an application uses compression or intricate caching mechanisms, the easiest way to improve content delivery speed is to *send less over the network*. With this in mind, a lot of research has been done with respect to protocols focused on eliminating redundant data transfers over HTTP links. Much of this redundancy is caused by the first-class nature of the 'file 'in traditional caching systems.

Inspired by research at Berkley on value based caching[4], Flying Squid's Fingerprinting system caches with respect to data, not file name. As a result, only changed and new parts of content need ever retransmitted over a network.

4.1 The API

Flying Squid's value based caching protocol takes the form of a simple, C++ class-based library. Modules exist on both the client and user side, and must be used together.

4.1.1 Client

```
class RabinClient
{
    public:
        /* Server name and port number*/
        RabinClient(char * hostname, int port_);

         ~RabinClient();

        /*
        * Receives a file from the server into the open, write enabled
            file pointer 'file'.
        * Returns the number of blocks added to the file.
         * This is a blocking call.
        */
        unsigned receive_file(FILE *file);

       /* Establishes a connection to the server */
        int connect_to_server();

         /* Disconnects from the server */
        int disconnect_from_server();

};
```

4.1.2 Server

```
class RabinServer
{

    public:
        /* Port at which to be open*/
        RabinServer(int port_);
        ~RabinServer();

        /* Sends a file of size s to the client.
```

```
            This is a blocking call.
        */
        int send_file(char *file, size_t s);


        /* Listens for the client and accepts a connection
        * Must be called before write_to_client is called */
        int connect_to_client();

         /* Connects to the client */
        int disconnect_from_client();

};
```

4.2 Integrating Value-Based Caching

Flying Squid uses modified TCP forwarding servers [3] at both the client and server sides to interface value-based caching with ATS.

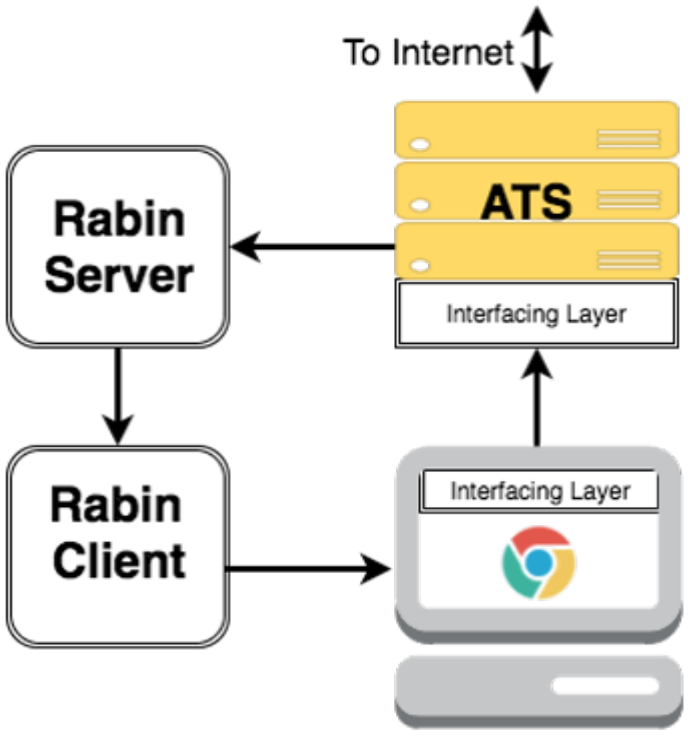

Figure 4: Flying Squid Value Based Caching Architecture

When the user makes a request through the browser, the browser forwards the request through its interfacing layer to the interfacing layer of ATS. ATS receives this request and gathers the requested data file from the Internet and the cloud caching mechanism previously described. ATS then sends over the file to the Rabin Server, which breaks down the file into blocks. The Rabin Server sends each block to the Rabin Client, which reassembles the blocks into a file. The Rabin Client then sends the response to the browser through the browser's interfacing layer. Lastly, the browser displays the response to the user.

4.3 Technical Approach

Flying Squid's Rabin Fingerprinting library provides a clean file transfer abstraction to the user, hiding the intricacies of a partial file transfer abstraction.

1. Server-side files are split into blocks of maximum size $1KB$ using a `djb-2`[5] based uniform random function of the form:
   $f : \{\text{byte} * \text{byte} * \text{byte}\} \rightarrow \{0, 1, ..., 1023\}$

2. Block boundaries are determined when this function returns 0. These blocks are treated as first-class objects and are hashed locally. A custom protocol can then be used to transfer the blocks representing a file over a TCP connection. Only the hash digests (identifiers) of previously transmitted blocks are transferred. Blocks of size 0 are used to denote EOF.

3. The client leverages the ordering properties of TCP to receive blocks in order. These are then locally stored, and identifiers are used to re-assemble them into traditional files.

4.4 Statistics

Pairs of files were generated that differ in the following ways:

1. Haskell: One file has an added byte.

2. HTML: Files differ by one large contiguous chunk.

3. C: Files differ by a large number of chunks distributed through the file

For each pair, these pairs of files were sent one after the other over a network using the Flying Squid fingerprinting system, with different maximum block sizes. Max block sizes smaller than 256 bytes were found to cause significantly more conflicts with the internal `djb2` hash.

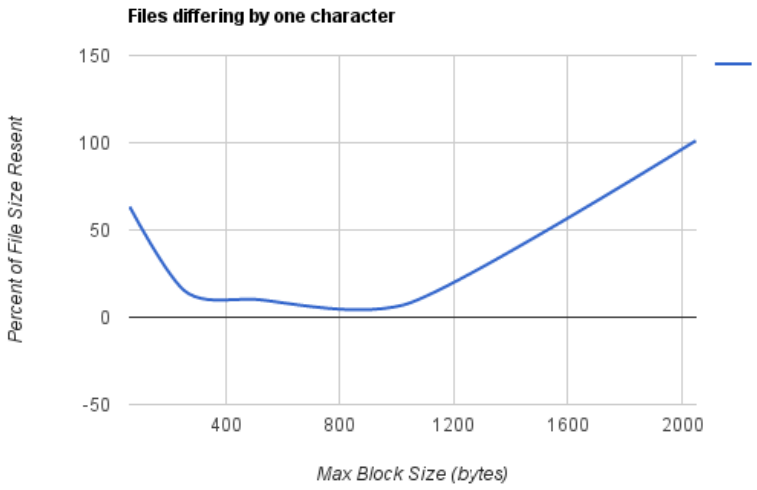

Figure 5: Details the percent of the second file sent at different max block sizes.

For files differing by one byte, a balance has to be struck between several small block headers being sent and the size of the retransmitted block(s). At most one block will be retransmitted due to the small difference between files. An optimal block size was found around 1000 bytes

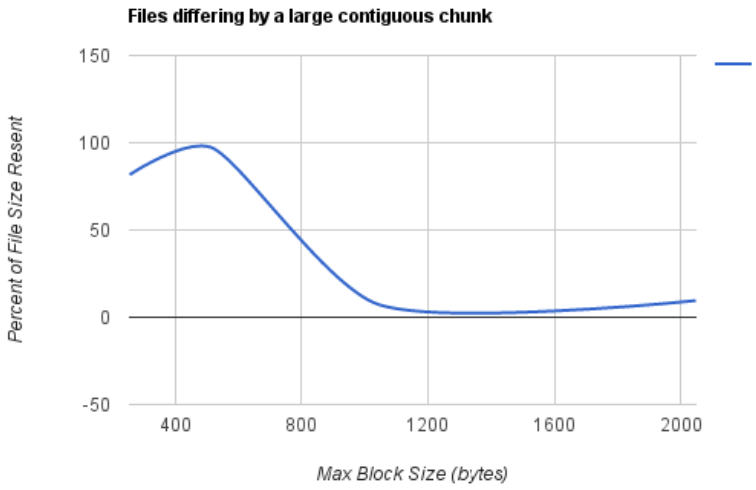

Figure 6: Details the percent of the second file sent at different max block sizes.

For files differing by a large contiguous chunk a minimum was found around 1400 bytes.

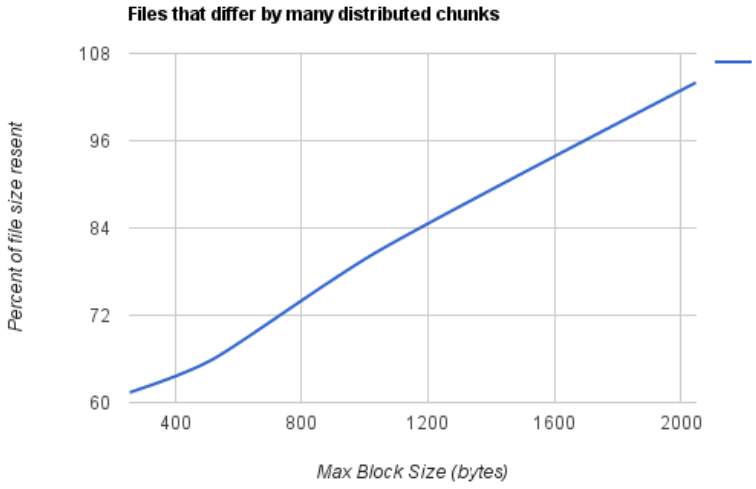

Figure 7: Details the percent of the second file sent at different max block sizes.

For files differing by a large number of chunks distributed through the file, smaller max block sizes were generally found to be better.

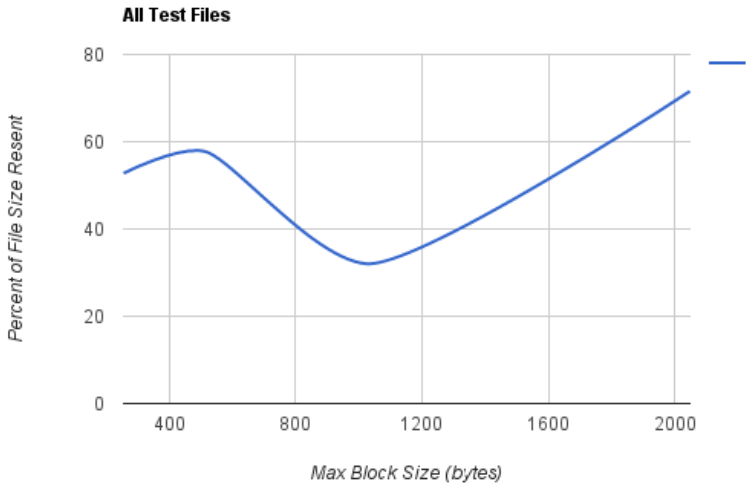

Figure 8: Details the percent of the second file sent at different max block sizes.

Taking the average of all these files, it was found that a minimal amount was resent around a maximum block size of $1KB$. As a result, the maximum value chosen was 1024 bytes, with an expected max size of 512 bytes.

5 Open Source Component

We have been working with an open source fork of ATS hosted in our GitHub organization. The modifications we have made to ATS will hopefully be proposed as pull requests we can make to the main project. Several of our contributions have been placed directly in the ATS codebase. Thus, all modifications are open source contributions that we hope to be approved as part of the main Traffic Server codebase.

The 'Rabin 'fingerprinting component of the project is also open source. It functions ias a standalone module that can be easily included in C++ code bases.

All of our GitHub repositories can be found at https://github.com/FlyingSquid/.

6 Issues and Challenges

7 Further Work

A lot of work needs to be done to make Flying Squid a production level proxy system. The proxy is only as strong as its weakest link. The integration of cloud caches, ATS and fingerprinting systems could be further streamlined. Better integration would involve more robust browser-side clients, and the elimination of redundant actions within ATS. Value based caching could be made more flexible, with adjustable block sizes and less persistent TCP connections. More benchmark and bandwidth analysis could help better understand the strengths and weaknesses of Flying Squid. Eventually, this analysis would provide grounds for interesting research.

To accurately measure the effectiveness and relevance of Flying Squid, we must investigate several use cases for our proxy server. We must leverage these use cases to show, with measurable benchmarks, that Flying Squid is indeed an improvement on the proxy status quo. First, there is the use case of a proxy network with multiple nodes to maximize personalized caching for more than just an individual. The custom caching could cater to a corporation, non-profit organization, as well as universities like Tufts. Many countries in the world have extremely limited bandwidth. We must account for these use cases, as these countries population constitute a significant part of the worlds internet users. In effect, Flying Squid will deliver content to these users at speeds faster than the speed that is currently available.

A User Manual

A.1 ATS with Cloud Caching

A.2 Value-Based Caching Integration

To run Flying Squid, run the following commands in the following order:

1. ```
   sudo /opt/ts/bin/trafficserver start
   ```

   ATS will be running on port 8080.

2. Set Google chrome or Mozilla Firefox to point to the port that the TCP Proxy Client (Browser's Interfacing Layer) will run on.

3. Start up the TCP Proxy Server (ATS's Interfacing Layer) in another terminal:

   ```
   cd $(RABIN_FINGERPRINT_DIR)/Integration/tcpproxy
   tcpproxy_server <local host ip> <tcpproxy_server port> <local host
       ip> 8080
   ```

4. Start up the TCP Proxy Client (Browser's Interfacing Layer) in another terminal:

   ```
   cd $(RABIN_FINGERPRINT_DIR)/Integration/tcpclient.
   tcpproxy_client <local host ip> <tcpproxy_client port> <local host
       ip> <tcpproxy_server port>
   ```

5. Open the browser that is pointing to the TCP Proxy Client and make request to a website, e.g. http://www.cs.tufts.edu.

A.3 Rabin Fingerprinting Demo

1. Start up the Rabin Fingerprinting Server:

   ```
   cd $(RABIN_FINGERPRINT_DIR)/Server
   ./rabinserver <Rabin Server port> ../Files/HTML/CSWebpage.html ../
       Files/HTML/CSWebpageWithoutHead.html
   ```

2. Start up the Rabin Fingerprinting Client in another terminal:

   ```
   cd $(RABIN_FINGERPRINT_DIR)/Client
   ./rabinclient localhost <Rabin Server port> test.html
   ```
   ```

B Better Integration: Google Native Client

Idea The idea behind using Google Native Client [2] is simply to eliminate the browser's interfacing layer. With Google Native Client, the browser can run C++ code in the browser. This way, the request does not have to pass through the TCP proxy interfacing layer and can go straight to ATS. Not only would this make Flying Squid more secure and portable, but it would also bring huge performance benefits.

In a web application, Google Native Client is embedded within the HTML through the use of an ¡embed¿ tag. The JavaScript and Google Native Client modules talk to each other via bidirectional, asynchronous messages.

Setting up Native Client Before Native Client can be set up, Python 2.7 and Make need to be available on the machine. Once these executables are available, the Native Client executable can be downloaded at the following website: https://developer.chrome.com/native-client/sdk/download. A Hello World Tutorial is also available at https://developer.chrome.com/native-client/devguide/tutorial/tutorial-part1.

For our purposes, however, perform the following steps to get up to speed:

1. Clone the clientInterfacing directory. The command is

   ```
   git clone https://github.com/FlyingSquid/clientInterfacing.git
   ```

2. Make sure your Google Chrome browser is version 49 or higher.
3. Change into the directory `nacl_sdk/pepper_49/getting_started/part1`.
4. Execute 'make '.
5. Change into the directory `nacl_sdk/pepper_49/getting_started`.
6. Execute 'make serve '.
7. Use the Google Chrome browser to access `http://localhost:5103`.

Debugging The application can also be opened on MAC OSX in the terminal via the following commands:

```
mkdir ~/Desktop/testtest

cd /Applications/Google Chrome.app/Contents/MacOS

./Google\ Chrome http://localhost:5103/part1 --enable-logging --v=1 --
    user-data-dir=~/Desktop/testtest
```

This will enable logging to appear, making it easier for future developers to debug the Native Client module.

Challenges Unfortunately, there were a few roadblocks that prevented Google Native Client from being a viable option for Flying squid at the moment. The roadblocks are listed below and are left for future developers to tackle.

1. Google Native Client cannot be compiled with g++ and the -std=c++0x flag, both of which are required to compile the Rabin Client and Rabin Server correctly.

2. Google Native Client cannot access the local filesystem of the machine that it is running on. It can only simulate a File I/O API using a ocal secure data store. This File I/O API is described at https://developer.chrome.com/native-client/devguide/coding/file-io. Further documentation of implementations of standard POSIX I/O functions such as fopen, fseek, fread, fwirte, and fclose are described at https://developer.chrome.com/native-client/devguide/coding/nacl_io.

References

[1] URL: https://trafficserver.readthedocs.org/en/latest/preface/index.en.html#what-is-apache-traffic-server (visited on 04/12/2016).

[2] URL: https://developer.chrome.com/native-client (visited on 05/03/2016).

[3] Arash Partow. *C++ TCP Proxy Server*. URL: http://www.partow.net/programming/tcpproxy/index.html (visited on 05/03/2016).

[4] Eric Brewer Sean C. Rhea Kevin Lian. "Value-Based Web Caching". In: *Proceedings of the 12th international conference on World Wide Web* (2003), pp. 619–628.

[5] Ozan Yigit. *Hash Functions*. URL: http://www.cse.yorku.ca/~oz/hash.html (visited on 04/06/2016).