Bosi Cheng
A53271697

Group member:
Zan Deng
A53285138

What are some of the challenges to delivering content over the Internet?

Peering point congestion, Inefficient routing protocols, Unreliable networks, Inefficient communications protocols, Scalability, Application limitations and slow rate of change adoption.

Why isn’t a simple web server model sufficient to efficiently deliver content? (In other words, what need or gap does Akamai address?)

Because a simple web server can face the challenges above but Akamai introduced the Content Delivery Network (CDN) concept to address these challenges.
Originally, CDNs improved website performance by caching static site content at the edge of the Internet, close to end users, in order to avoid middle mile bottlenecks as much as possible. Since then the technology has rapidly evolved beyond static web content delivery. Afterwards, Akamai had application delivery networks that can accelerate entire web or IP-based applications, media delivery networks that provide HD-quality delivery of live and on-demand media, and EdgeComputing networks that deploy and execute entire Java J2EE applications in a distributed fashion

Briefly describe how Akamai enables distribution of live and streaming content?

1. Through CDN: Afterwards, Akamai had application delivery networks that can accelerate entire web or IP-based applications, media delivery networks that provide HD-quality delivery of live and on-demand media, and EdgeComputing networks that deploy and execute entire Java J2EE applications in a distributed fashion

2. The real-time mapping: The real-time mapping part of the system creates the actual maps used by the Akamai platform to direct end users (identified by the IP addresses of end users and their name servers) to the best Akamai edge servers to respond to their requests. This part of the system also selects intermediates for tiered distribution and the overlay network.

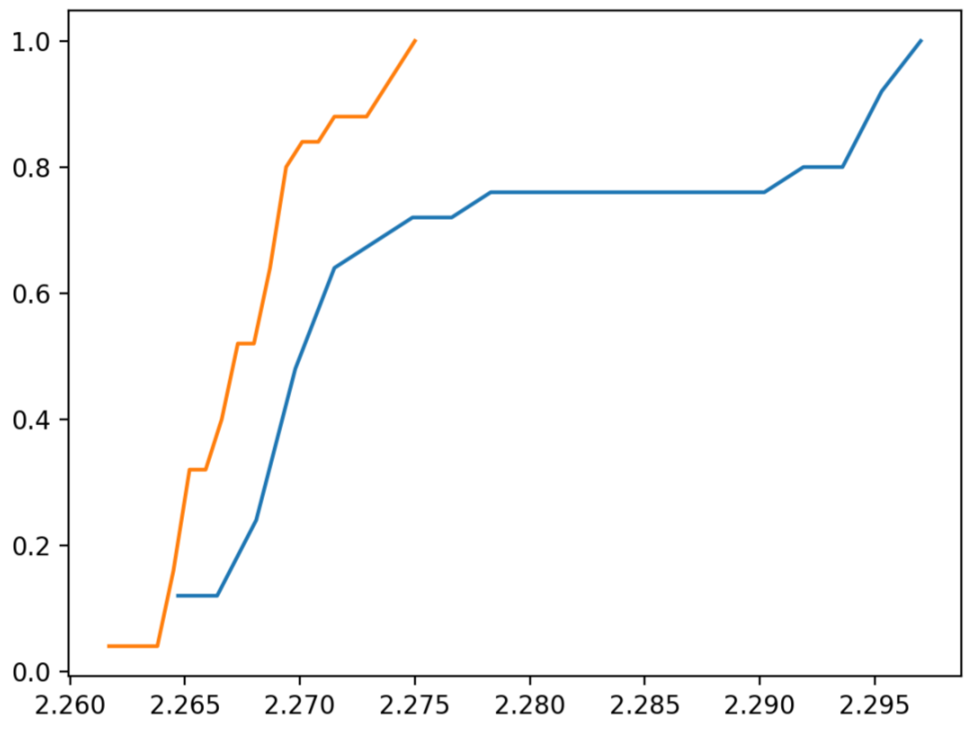
What information does the Akamai CDN use to map requests to resources? How (briefly) does it implement that mapping?

Historic and real-time data.
The real-time mapping part of the system creates the actual maps used by the Akamai platform to direct end users (identified by the IP addresses of end users and their name servers) to the best Akamai edge servers to respond to their requests. This part of the system also selects intermediates for tiered distribution and the overlay network. This assignment happens in two main steps: Map to cluster and Map to server.

When hardware and network faults are identified (such as a failing hard drive on a server), the failed edge servers are ―suspended and will finish up processing in-progress requests but will not be sent any additional end users until the fault has been resolved.

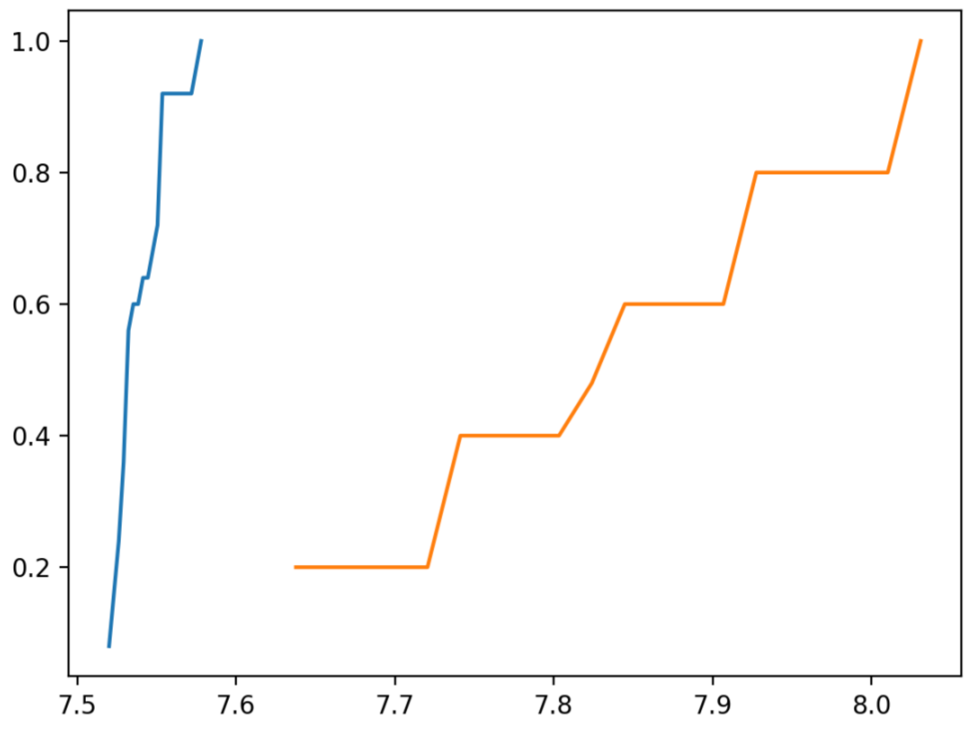
The mapping system itself is a fault-tolerant distributed platform that is able to survive failures of multiple data centers without interruption. The scoring and map-to-cluster portions of the system run in multiple independent sites and leader-elect based on the current health status of each site. The map-to-server portions of the system run on multiple machines within each target cluster. All portions of the system, including monitoring agents and DNS servers, communicate via a multi-path overlay transport that is able to tolerate network faults.

small files:



The orange plot is the "Nearest to client" algorithm and the blue is the original one.
Obviously the second algorithm performs better.
The average difference is 0.00764.

large files:



The orange plot is the "Nearest to client" algorithm and the blue is the original one.
Obviously the second algorithm performs better.
The average difference is 0.2858.

conclusions:

The file size does matter. While applying the original algorithm the chances are the client store some of the blocks to a blockstore which takes longer time to respond (longer ping). However, the second algorithm can guarantee that the client always picks the blockstore which takes least time to respond.

Besides, the larger the files are, the bigger the difference between two algorithms is. The larger file has more blocks to store and as long as there is even one block stored not at the best blockstore it takes longer time.

client.py:

```
import rpyc

import hashlib

import os

import sys

from metastore import ErrorResponse

import time



"""

A client is a program that interacts with SurfStore. It is used to create,

modify, read, and delete files.   Your client will call the various file

modification/creation/deletion RPC calls.   We will be testing your service with

our own client, and your client with instrumented versions of our service.

"""



class SurfStoreClient():

    """

    Initialize the client and set up connections to the block stores and

    metadata store using the config file

    """
```

```
def __init__(self, config):
    config_file = open(config, 'r')
    arr = []
    for line in config_file.readlines():
        arr.append(line)
    config_file.close()
    self.numBlockStores=arr[0].split(": ")[1]
    self.metastore=rpyc.connect(arr[1].split(": ")[1].split(":")[0],arr[1].split(": ")[1].split(":")[1].strip())
    self.blockstore=[]
    for line in arr[2:]:
        if line.strip() == "" or line == "\r\n":
            continue
        block_connect=rpyc.connect(line.split(": ")[1].split(":")[0], line.split(": ")[1].split(":")[1].strip())
        self.blockstore.append(block_connect)


"""
upload(filepath) : Reads the local file, creates a set of
hashed blocks and uploads them onto the MetadataStore
(and potentially the BlockStore if they were not already present there).
```

```
"""



def upload(self, filepath):

    blockstore_index=self.findServer()

    conn = self.metastore

    file_name = filepath.split("/")[-1]

    (v,hl,bl) = conn.root.read_file(file_name)

    upload_file = open(filepath, 'rb')

    buffer = b''

    count = 0

    upload_hl = []

    upload_dict = {}

    for byte in upload_file:

        if count != 0 and count % 4096 == 0:

            hash_value=hashlib.sha256(buffer).hexdigest()

            upload_hl.append(hash_value)

            upload_dict[hash_value]=buffer

            buffer = b''

        buffer += byte

        count += 1
```

```python
        hash_value=hashlib.sha256(buffer).hexdigest()

        upload_hl.append(hash_value)

        upload_dict[hash_value]=buffer

        try:

            conn.root.modify_file(file_name, v + 1, upload_hl,blockstore_index)

        except Exception as e:

            if e.error_type==2:

                self.upload(filepath)

            if e.error_type == 1:

                for value in eval(e.missing_blocks_list):

                    self.blockstore[blockstore_index].root.store_block(value,upload_dict[value])

        print("OK")




    """

    delete(filename) : Signals the MetadataStore to delete a file.

    """


    def delete(self, filename):
        conn = self.metastore

        (v,hl,blockstore_index) = conn.root.read_file(filename)

        if v==0:

            print("Not Found")

            return -1

        try:

            conn.root.delete_file(filename, v + 1)

        except Exception as e:

            if e.error_type==2:

                self.delete(filename)

        print('OK')





    # check if succeed



    """

        download(filename, dst) : Downloads a file (f) from SurfStore and saves

        it to (dst) folder. Ensures not to download unnecessary blocks.

    """
```

```
def download(self, filename, location):
    (v,hl,blockstore_index)=self.metastore.root.read_file(filename)
    # print(v,hl)
    if len(hl)==0:
        print("Not Found")
        return -1
    else:
        content=b''
        for h in hl:
            # blockstore_index=self.findServer()
            buffer=self.blockstore[blockstore_index].root.get_block(h)
            content+=buffer

    filepath=location+"/"+filename
    file=open(filepath,'wb')
    file.write(content)
    file.close()
    print('OK')
    def findServer(self):

        blockstore_index=0

        min_RTT=sys.maxsize

        for i in range(len(self.blockstore)):

            send_out=time.time()

            self.blockstore[i].ping()

            recv=time.time()

            RTT=round(recv-send_out,4)

            if RTT<min_RTT:

                blockstore_index=i

                min_RTT=RTT

        return blockstore_index





if __name__ == '__main__':

    client = SurfStoreClient(sys.argv[1])

    operation = sys.argv[2]

    if operation == 'upload':

        client.upload(sys.argv[3])
```

```python
elif operation == 'download':

    client.download(sys.argv[3], sys.argv[4])

elif operation == 'delete':

    client.delete(sys.argv[3])

else:

    print("Invalid operation")
```

metastore.py:

```
import rpyc

import sys

from collections import *



'''

A sample ErrorResponse class. Use this to respond to client requests when the request has any of the

following issues -

1. The file being modified has missing blocks in the block store.

2. The file being read/deleted does not exist.

3. The request for modifying/deleting a file has the wrong file version.



You can use this class as it is or come up with your own implementation.

'''




class ErrorResponse(Exception):

    def __init__(self, message):

        super(ErrorResponse, self).__init__(message)

        self.error = message
    def missing_blocks(self,hashlist):

        self.error_type = 1

        self.missing_blocks_list = hashlist



    def wrong_version_error(self, version):

        self.error_type = 2

        self.current_version = version


    def file_not_found(self):

        self.error_type = 3




'''

The MetadataStore RPC server class.


The MetadataStore process maintains the mapping of filenames to hashlists. All

metadata is stored in memory, and no database systems or files will be used to

maintain the data.

'''
```

```
class MetadataStore(rpyc.Service):

    """

        Initialize the class using the config file provided and also initialize

        any datastructures you may need.

    """



    def __init__(self, config):

        config_file = open(config, 'r')

        arr = []

        for line in config_file.readlines():

            arr.append(line)

        config_file.close()

        self.numBlockStores=arr[0].split(": ")[1]

        self.blockstore=[]

        for line in arr[2:]:

            if line.strip() == "" or line == "\r\n":

                continue

            block_connect=rpyc.connect(line.split(": ")[1].split(":")[0], line.split(": ")[1].split(":")[1].strip())

            self.blockstore.append(block_connect)

        self.file_map = defaultdict(list)
```

```
'''
    ModifyFile(f,v,hl): Modifies file f so that it now contains the
    contents refered to by the hashlist hl.   The version provided, v, must
    be exactly one larger than the current version that the MetadataStore
    maintains.

    As per rpyc syntax, adding the prefix 'exposed_' will expose this
    method as an RPC call
'''

def exposed_modify_file(self, filename, version, hashlist,blockstore_index):
    # (v,hl)=self.exposed_read_file(filename)

    mbl = []
    for hash_value in hashlist:
        # blockstore_index=self.findServer(hash_value)
        if not self.blockstore[blockstore_index].root.has_block(hash_value):
            mbl.append(hash_value)
        if mbl==[] and self.file_map[filename][1]!=[]:

            return "OK"

        else:

            if version <= self.file_map[filename][0]:

                error = ErrorResponse("wrong_version_error")

                error.wrong_version_error(self.file_map[filename][0])

                raise error

            self.file_map[filename] = [version, list(hashlist),blockstore_index]

            error = ErrorResponse("missing_blocks")

            error.missing_blocks(mbl)

            raise error





    '''

        DeleteFile(f,v): Deletes file f. Like ModifyFile(), the provided

        version number v must be one bigger than the most up-date-date version.


        As per rpyc syntax, adding the prefix 'exposed_' will expose this

        method as an RPC call
```

```
'''



# def exposed_delete_file(self, filename, version):

#       if version<=self.file_map[filename][0]:

#               error = ErrorResponse("wrong_version_error")

#               error.wrong_version_error(self.file_map[filename][0])

#               raise error

#       [(curr_v,hl)]=self.file_map[filename]

#       self.file_map[filename]=(curr_v+1,[])





'''

    (v,hl) = ReadFile(f): Reads the file with filename f, returning the

    most up-to-date version number v, and the corresponding hashlist hl. If

    the file does not exist, v will be 0.



    As per rpyc syntax, adding the prefix 'exposed_' will expose this

    method as an RPC call

'''
```

```
    def exposed_read_file(self, filename):

        if filename not in self.file_map.keys():

            self.file_map[filename]=[0,[],-1]

        [v,hl,bl]=self.file_map[filename]

        return (v,hl,bl)




    # def findServer(self,h):

    #        return int(h,16) % int(self.numBlockStores)



if __name__ == '__main__':

    from rpyc.utils.server import ThreadedServer



    server = ThreadedServer(MetadataStore(sys.argv[1]), port=6000)

    server.start()
```

blockstore.py:

```
import rpyc
import sys



class BlockStore(rpyc.Service):
    """
    Initialize any datastructures you may need.
    """

    def __init__(self):
        self.block_map = {}
    """
        store_block(h, b) : Stores block b in the key-value store, indexed by
        hash value h

        As per rpyc syntax, adding the prefix 'exposed_' will expose this
        method as an RPC call
    """
```

```
def exposed_store_block(self, h, block):

    self.block_map[h] = block



"""

b = get_block(h) : Retrieves a block indexed by hash value h



    As per rpyc syntax, adding the prefix 'exposed_' will expose this

    method as an RPC call

"""



def exposed_get_block(self, h):

    return self.block_map[h]



"""

    rue/False = has_block(h) : Signals whether block indexed by h exists

    in the BlockStore service



    As per rpyc syntax, adding the prefix 'exposed_' will expose this

    method as an RPC call

"""
```

```
    def exposed_has_block(self, h):

        if h in self.block_map.keys():

            return True

        else:

            return False



if __name__ == '__main__':

    from rpyc.utils.server import ThreadPoolServer

    port = int(sys.argv[1])

    server = ThreadPoolServer(BlockStore(), port=port)

    server.start()
```