IN4144 - DataScience 2014 report

by Wing Lung Ngai and Sietse T. Au
31-08-2014

Proposed idea

A web archive such as archive.org would be more suitable for such things. However the data is not open to the public, apart from researchers, scholars etc. This notion makes a difference, but we'll get back to this later. To construct a timeline, it's possible to only use the CommonCrawl snapshots of the web, as they are timestamped. Currently there are snapshots of ~2008 - 2014 available; the granularity is a bit coarse. On the NorvigAward website we can see that the only the 2014 set is available. We can augment the 2014 data using the JSON-API of the Wayback machine on archive.org. As for the security analysis we can use data from http://www.cvedetails.com/. Depending on the time allocated for this lab relative to the entire course we can decide which type of security to focus on.

Methodology

In this section we will elaborate the choices made during the execution of the project.

Choosing a large scale data processing platform

In the IN4144 data science course, we were recommended to use Hadoop MapReduce or Pig to produce MapReduce programs for experimenting with the proposed idea. To be able to execute this program on the available data set, access was given to the SURFsara Hadoop cluster. On the SURFsara website we found the following statement: “The software we run on the cluster is the Cloudera CDH3 distribution with Apache Hadoop 0.20.205.”, which would restrict us to Hadoop MapReduce programs. However, Apache Hadoop 2.4.0is deployed on the SURFsara cluster. The 2.x Apache Hadoop release is called YARN, which instead of running only MapReduce programs, will run any arbitrary program if one would write an ApplicationMaster for it. In the past year many data processing platforms have created deployment scripts for YARN.

We set our eyes on Apache Spark, which uses Resilient Distributed Dataset as an immutable data abstraction. Unlike Hadoop MapReduce, the result is not read/written to HDFS with step, but instead to memory, better performance can be achieved. The promise is 10 - 100 times higher performance than Hadoop MapReduce.

WARC format

For our idea, we need the HTTP response headers, these are only found in the WARC files. Spark has an interface for using Hadoop InputFormat. Thus the provided WarcInputFormat class from SURFsara warcutils is used to parse raw war.gz file.

Distributed computation

The Spark data distribution to workers is by default done by looking at the number of blocks a file is split in. For example, if a file is is 1024MB and the block size is 128MB, then 8 workers will be employed for computation. This partitioning can be manipulated by the programmer.

Environmental set up

The SURFsara Hadoop cluster consists of 90 compute nodes each with 8 cores and 64GB RAM. It uses Kerberos for authentication. In this section, we elaborate on how we develop for Apache Spark and how we deploy the Apache Spark application on the the SURFsara Hadoop 2.4.0 cluster.

Development environment

SBT 0.13 is used for building jars for Spark applications with the main language Scala 2.10.4 for Java 7, using the following libraries, org.apache.spark/spark-core_2.10/1.0.2, SURFsara/warcutils/2.1 and org.jwat/jwat-warc/1.0.1. (groupId/artifactId/version)

Job submission environment

An environment for job submission is provided in the following GitHub repository: https://github.com/sara-nl/hathi-client . With this, it is possible to communicate with the Hadoop cluster.

Apache Spark 1.0.2 is used for the experiments, as of August 5th 2014, this is the latest version. Spark has an interface to submit jobs to YARN clusters. This interface is a bash script called “spark-submit”. Through it we specify

- master
    - This is the type of master, because we are deploying on a YARN cluster we set this parameter to “yarn-cluster”. It is also possible to set this parameter to “yarn-client”, this means that the master will be run on your local machine (which may not be ideal)
- driver-memory
    - This is the memory available to the master
- executor-memory
    - The memory available to a single worker
- executor-cores
    - The number of cores in a single worker
- num-executors
    - The total number of workers
- class
    - The class to be ran
- jar
    - The jar which contains the class
- jar_args
    - The optional arguments to the class

To be able to submit to the SURFsara Hadoop cluster we have to export the environment variable **HADOOP_CONF_DIR** which points to the hadoop configuration in the hathi-client.
The final configuration used to set-up Spark "spark-defaults.conf":

```
spark.serializer        org.apache.spark.serializer.KryoSerializer
spark.reducer.maxMbInFlight        100
spark.kryoserializer.buffer.mb     100
spark.akka.frameSize 200
spark.storage.memoryFraction 0
```

- spark.serializer - The standard Java serializer is extremely slow, Kryo is a much faster serializer
- spark.reducer.maxMbInFlight - increased due to fails
- spark.kyroserializer.buffer.mb - increased due to fails
- spark.akka.frameSize - increase due to communication fails
- spark.storage.memoryFraction - set from 0.6 to 0, all memory is now used for processing data and none is reserved for caching.

Extracting data from the MAIN dataset

In this section, the way in which we extract the data will be elaborated.

Transforming the MAIN dataset

The idea is to eventually construct a timeline, such that we will be able to gauge the relative vulnerability of the public world wide web. We limit ourselves to examining the following HTTP response headers: **Server**, **X-Powered-By**, **X-AspNet-Version, X-Runtime, X-Version**.
To do this, we only need the WARC records. For each WARC record, we check whether it is a response by looking at the **header.warcTypeIdx** field, and only get the domains which have a valid **header.warcTargetUriUri**. The aforementioned headers are then parsed into a headerKey -> headerValue map for each WARC record, the host of the **warcTargetUriUri** is extracted and we have a timestamp from the **header.warcDate**. Thus for each WARC record we have the following object **{host, {date, headerKey -> headerValue map}}**. The hosts are then grouped, such that we obtain per host: **{host, list of {timestamp, headerKey -> headerValue maps}}**. Subsequently this is serialized to HDFS for reuse. By using this extracted dataset, we save a lot of time decompressing, because the main dataset is gzip compressed.

MAIN dataset

Using the following HDFS command we get the size of the datasets:

```
hdfs dfs -du -s /path/of/file/or/directory
```

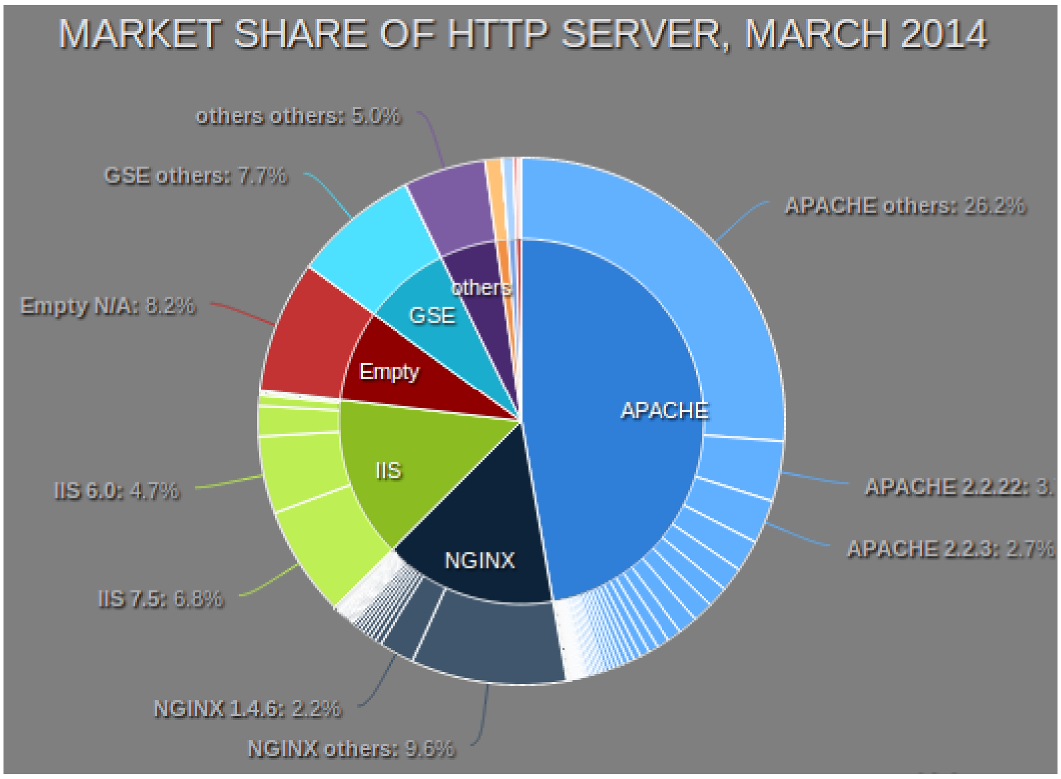
The main dataset on the cluster /data/public/common-crawl/crawl-data/CC-MAIN-2014-10 (March 2014) has a size of 85864335485586 bytes or 85.8TB. The test dataset /data/public/common-crawl/crawl-data/CC-TEST-2014-10 has a size of 153057493582 bytes or 153GB. These sizes contradict the sizes on http://norvigaward.github.io/faq.html#datasets namely 48.6TB for the MAIN dataset and 80GB for the TEST dataset. Or the size given on the CommonCrawl website, which states that the dataset is 223TB.

Results

The dataset we have extracted from the MAIN dataset has a size of 202026911608 bytes or 202GB. The number of “unique” domains crawled is 249965073, however this count does not make a difference between subdomains or top-level or second-level domains. For example “x.blogspot.nl”, “x1.blogspot.nl” are counted as 2. According to
http://techcrunch.com/2013/04/08/internet-passes-250m-registered-top-level-domain-names/
, the total number of domains registered hit 252m in April 2013. Quantity wise, it is close to the real number of registered domains, however we do not know what the real number of domains of the CommonCrawl dataset is.

Marketshare of HTTP Servers

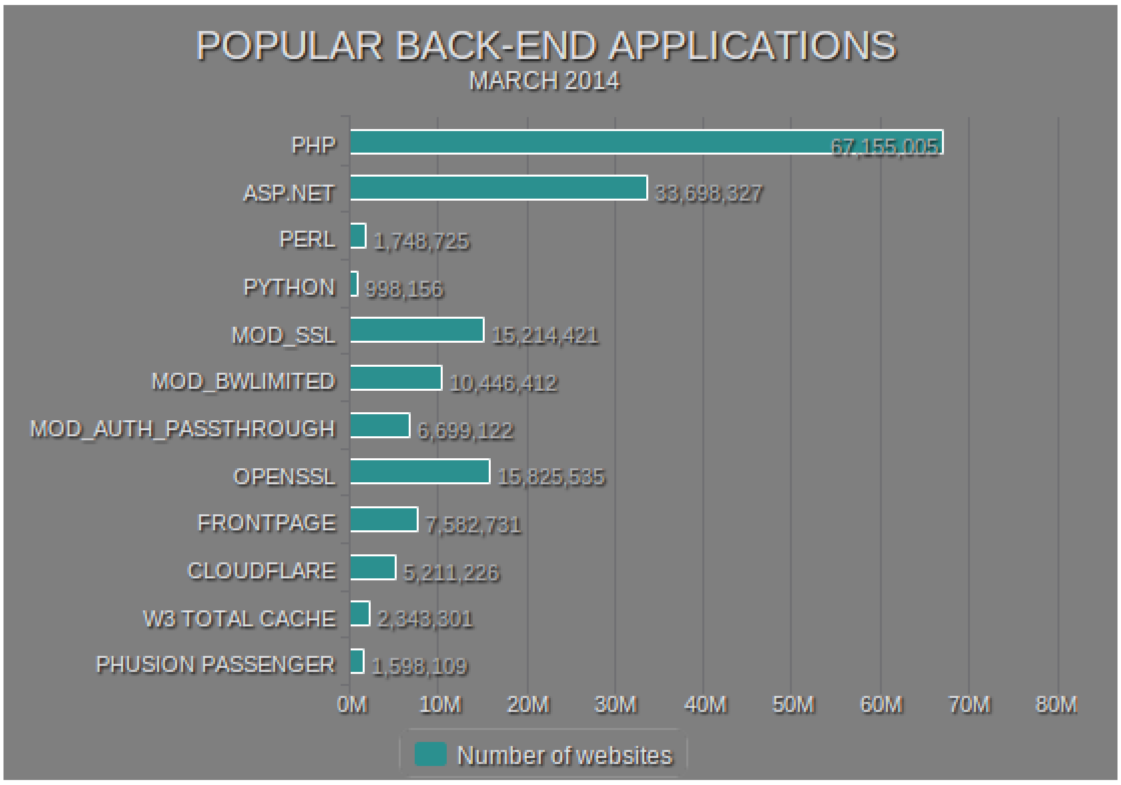
First we wanted to see the marketshare of different HTTP servers. This way we can reduce the look-ups to the CVEDetails, if we find that the market is dominated by few players.



From 249,965,073 extracted web domains, the top 3 most popular servers, namely Apache(47.2%), Nginx(15.3%) and IIS(14.3%) have more than 75% of the market share. 8.2% of the web domains didn't return their server information. Sometimes the web domains also provided version information.

Marketshare of back-end software

Besides the HTTP servers, other related back-end web applications and languages used by more than 1 million web domains are extracted from the dataset.



Relating it to vulnerability

To properly assess the vulnerability we'd need to look at the worst valued bug for each version of each server platform or execution engine. However due to time constraints we'll look at the weighted average CVSS score (scale 1 - 10) of the past year for the top 3 in the server platforms.

- Apache HTTP-Server: 5.8
- nginx: 0
- IIS: 7

For the back-end software packages

- PHP: 5.8
- ASP.NET: 0
- OpenSSL 5.7
- Perl: 5
- Python: 6.5
- Phusion passenger: 5

Discussion

Partial execution of the original idea

Initially the aim was to construct a timeline of web vulnerability, however it took much more time to deploy and tweak the settings of the Spark platform (to enable the jobs to not fail).

Too many tasks

The master will not be able to cope with the administration of a lot of tasks, at some point the master will freeze and the job will fail. The solution to this is to manually adjust the task parallelism to a lower number.

Payload and communication failure

Spark uses the Akka framework for the communication, where arbitrarily sized payloads are sent over the network, however the framesize in which a payload must fit is static. We had to adjust this framesize manually to allow the jobs to finish.

Caching in a multi-user environment

One of Spark’s key points is that it can cache datasets for re-use in memory, however in a multi-user environment such as a shared Hadoop cluster. The memory bandwidth may be shared with other users, there’s also serialization and deserialization overhead. In our case caching was a hard hit on the performance, we decided to not use caching at all.

Out of memory exception

Both the master and worker can run out of memory. A master managing a lot of small tasks may trigger the garbage collection which will freeze the master, once the master is too busy garbage collecting, the job will fail due to workers timing out.
The other way is also possible, the workers freeze due to heavy garbage collection, there’ll be a time-out and your job may or may not fail. To solve this, we had to increase the memory of the master and worker for our aggregation jobs.

Future work

Constructing the timeline by sampling

By using the unique domains, we can sample a subset of these domains to make HTTP requests to get a current view of the web. This same subset can be used to query the WaybackMachine using the Memento API.
Then by analyzing the results as described in the previous sections of this report we will be able to see the evolution of the vulnerabilit of the WWW governed by the moving market shares of different platforms.