# Go based content filtering software on FreeBSD (Developing a content filtering software in Go on FreeBSD)

Ganbold Tsagaankhuu, *Mongolian Unix User Group,*
ganbold@gmail.com
Esbold Unurkhaan, *Mongolian University of Science and Technology*
esbold@must.edu.mn
Erdenebat Gantumur, *Mongolian Unix User Group*
erdenebat.gantumur@gmail.com

**Abstract**—Go is a new programming language, compared to many other programming languages like C, C++, Java, etc., but it has many practical features and is in many cases more productive. On the other hand, FreeBSD has been around for very long time and proven to be the most reliable, one of the most powerful operating system available today. In this paper, we will discuss the issues, pros, cons, and common pitfalls of developing software in Go on FreeBSD. We chose content filtering software for this purpose and called our project Shuultuur. Shuultuur is a Mongolian word, which means "filter" in English. First, we will describe the rational behind our choices for setting up our development environment and toolchain. In addition, we will list specific hurdles that we faced in regards to content filtering software, Go and FreeBSD. Furthermore, our real world benchmarking results in contrast to Dansguardian and other findings will be presented. Finally, we will conclude and discuss possible future works.

**Keywords**—Content filter, String matching, Go language, FreeBSD

## 1 Introduction

In our everyday life, we are witnessing how the modern world is moving towards the Internet of Things and the connected world is expanding quickly. This emerging change made us to look into existing problems from a different prospective. For us, one of these problems was content filtering. Therefore, we decided to take a journey to pursue our idea. There are countless numbers of open source projects, which are backed by various communities available today. However, some open source projects didnt evolve well enough throughout their lifetime. Therefore, it became difficult to improve existing code base because of various reasons. Since our content filtering project idea grew out of specific needs, we decided to develop it from the scratch. Therefore, in order to save years of development time, we needed to be productive to match up with mature content filtering softwares features and performance, and take it further. These initial requirements made us search for something different.

## 2 Rationale behind our choices

### 2.1 Why content filter?

First of all, in regard to the main objective of content filtering, the common understanding is to have some sort of control over unwanted content from web content. These types of solutions are widely used in enterprises to enforce their computer security policies. The public organizations such as libraries, schools, etc. use content filters to protect children from content inappropriate for their age i.e. adult, violence, drugs etc. Therefore, content filtering can perform wide variety of tasks and we believe that there is a specific need of content filtering.

### 2.2 Programming Language Choice

One of the questions that we had in mind was which programming language to use? We have looked around and considered a number of popular programming languages. Essentially, we were looking for a programming language that is fast, lightweight, easy to prototype, and that requires relatively minimal effort to produce and maintain production quality code. Therefore, we preferred a statically typed, compiled language with strong type system. Of course, there is always a tradeoff, however, when it comes to the question of achieving our goal faster, the previously mentioned characteristics made sense for our project.

Go is relatively new programming language and it was officially launched at Google 5 years ago [1]. Go is compiled, statically typed, garbage collected, unconventionally object oriented and general-purpose system programming language. Go produces native binaries, has a very fast compilation time and is designed with concurrency in mind. Therefore, since Go's characteristics matched with our initial requirements well, we looked closely into it along with other candidates and started experimenting with it. Performance of Go's native binaries was somewhat comparable to good old low-level C and tremendously better than interpreted languages [5]. After extensive research and trials, we concluded that Go programming language is the best suited for our goal and we will briefly give our reasons below.

Go did not need any additional library to deal with concurrency; it is already part of the programming language features and it has strong support for multiprocessing. In addition, Go is a more productive language compared to C and includes multiple useful built-in data structures such as maps [23] and slices [24]. Especially when dealing with concurrency, many advanced practical solutions can be easily used in modern hardware using Go specific features such as goroutines [21] and channels. A goroutine is a function executing concurrently with other goroutines in the same address space. It is lightweight and communicates with other goroutines via channels [22]. Because Go is very simple, garbage collected and statically typed language in nature, source code can be written with less errors and mistakes, thus presumably with less bugs. Furthermore, it is relatively easy to profile for speed and memory leaks that is handy when working on production source code. In terms of syntax, it has loosely derived from C and influenced by other languages such as Python [3]. Also, Go has an extensive number of libraries [2] and finally, Go is a BSD licensed completely open source language [4].

In the context of content filtering, to detect the meaning of a sentence accurately is a hard task for an automated tool. As we know, humans cannot detect the real meaning of sentences without detailed information. However, in the real world, content filters try to classify web contents based on string matching techniques into bad content, which should be blocked or good content, which should be allowed [6]. In most languages, the exact string matching technique (pattern matching) may demand high processing power [7]. For this reason, we chose to develop content filtering software to take the advantage of Go's performance.

### 2.3 Why FreeBSD as a platform of choice:

We have chosen FreeBSD OS as a main development and testing platform mainly because:

- It is one of the most powerful, mature and stable operating systems as well as a complete, reliable, self-consistent distribution.
- FreeBSD's networking stack is very solid and fast [8].
- One of the advantages of choosing FreeBSD is its port and package system, which makes it easy to install and deploy the necessary applications and software.
- Handy tools such as NanoBSD exist, which can be used to make custom FreeBSD image easily.
- Finally, we love FreeBSD.

## 3 Related projects

We used the following open source software for our project:

- goproxy provides a customizable HTTP proxy library for Go. It supports regular HTTP proxy, HTTPS through CONNECT, and "hijacking" HTTPS connection using "Man in the Middle" style attack. The intent of the proxy is to be usable with reasonable amount of traffic yet, customizable and programmable [9].
- gcvis - Visualizes Go program gctrace data in real time [10].
- profile is a simple profiling support package for Go [11].
- go-nude is nudity detection with Go [12].
- xxhash-go is a go wrapper for C xxhash - an extremely fast Hash algorithm, working at speeds close to RAM limits [13].
- powerwalk is a Go package for walking files and concurrently calling user code to handle each file [14].
- redigo is a Go client for the Redis database [15].
- Redis. It is open source, BSD licensed, advanced key-value cache and store [16].

## 4 Experienced challenges

We have faced several problems during development that are listed below:

- The Shallalist blacklist contains more than 1.8 million URL/Domain entries. Storing

```
# go tool pprof --alloc_space ./shuultuur_mem /tmp/profile228392328/mem.pprof
Adjusting heap profiles for 1-in-4096 sampling rate
Welcome to pprof!  For help, type 'help'.
(pprof) top15
Total: 11793.7 MB
  3557.7  30.2%  30.2%   3557.7  30.2% runtime.convT2E
  1212.1  10.3%  40.4%   1212.1  10.3% container/list.(*List).insertValue
   832.3   7.1%  47.5%   2434.8  20.6% github.com/garyburd/redigo/redis.(*conn).readReply
   807.9   6.9%  54.4%   1874.6  15.9% github.com/garyburd/redigo/redis.(*Pool).Get
   673.8   5.7%  60.1%    673.8   5.7% github.com/garyburd/redigo/redis.Strings
   544.5   4.6%  64.7%    549.4   4.7% main.regexBannedWordsGo}
   521.1   4.4%  69.1%    521.1   4.4% bufio.NewReaderSize
   490.9   4.2%  73.3%    490.9   4.2% bufio.NewWriter
   438.2   3.7%  77.0%    438.2   3.7% runtime.convT2I
***  369.8   3.1%  80.1%   7622.9  64.6% main.workerWeighted
   255.0   2.2%  82.3%    255.9   2.2% main.regexWeightedWordsGo
   235.5   2.0%  84.3%    235.5   2.0% bytes.makeSlice
   229.9   1.9%  86.2%    397.1   3.4% io.Copy
   168.3   1.4%  87.6%    168.3   1.4% github.com/garyburd/redigo/redis.String
***  162.6   1.4%  89.0%   4048.9  34.3% main.getHkeysLen
(pprof)
```

*Fig. 1: Pprof result - Memory allocation in initial stage*

them in memory was challenging and initially we stored the URL/Domain entries in Redis in the following way:

```
...
// Store URL/Domains as a key and
// category as value
conn.Do("SET", urls_or_domain, category)
...
```

This was not effective in terms of memory utilization and performance. After a bit of research, we have found a way to reduce it to around 4100 hash keys. We used Stephane Bunel's xxhash-go to compute a hash from each URL/Domain and sliced it and then stored those slices in Redis similar to the following way:

```
...
// use xxhash to get checksum from URL/Domain
blob := []byte(url_or_domain)
h32g := xxh.GoChecksum32(blob)
/*
* Store it as hash in Redis in following way:
*   key    = 0xXXXX (first half of URL/Domain),
*   field  = XXXX (second half of URL/Domain),
*   value = category
*/
hash_str := fmt.Sprintf("0x%08x", h32g)
key        := hash_str[0:6]
value      := hash_str[6:]
conn.Do("HSET", key, value, category)
...
```

- Banned and weighted phrase lookup problem. Originally they were stored in Redis, and accessing them in a loop was slow and inefficient. We improved it using a graph and map. An every word that exists in the phrase lists (banned, weighted etc.) is an edge of the graph and it should be unique in the same category. For example, we have banned phrases such as ″sex woman″, ″sex man″ and ″mature sex″. Shuultuur creates four Edges such as ″sex″, ″woman″ , ″man″ and ″mature″ and Vertices. The Edges and their related Vertices are stored in the map because of programming efficiencies. Go language provides a built-in map type that implements a hash table. In addition, we have replaced a regular expression based search algorithm with Boyer Moore search algorithm, which is implemented in Go.
- Reading HTTP response bodies into strings makes the heap memory usage grow very large due to lots of allocations, especially when the rate of connections per second is high. Ideally, this should be processed using a streaming parser by utilizing the io.Reader interface. Also, limiting the connection rate on incoming requests could be an option. We have optimized and improved it by doing some CPU and memory profiling [19]. This is done by enabling memory profiling in Shuultuur and we have used Go's built-in profiler pprof. The Figure 1 shows report in memory allocations during the initial stage of development:
In this initial report you can see lots of allocations in *main.workerWeighted* and *main.getHkeysLen*. Those functions were used for searching banned and weighted phrases using Redis. We improved Shuultuur by removing those functions, did some code level optimizations and introduced a better algorithm. The Figure 2 is the report generated by the same command after the previously mentioned improvements were done and we think that there is still some room for further improvements.

```
# go tool pprof --alloc_space ./shuultuur /tmp/profile287823990/mem.pprof
Adjusting heap profiles for 1-in-4096 sampling rate
Welcome to pprof!  For help, type 'help'.
(pprof) top30
Total: 2156.3 MB
   596.9  27.7%  27.7%   1066.4  49.5% io.Copy
   406.3  18.8%  46.5%    406.3  18.8% compress/flate.NewReader
   177.3   8.2%  54.7%    177.4   8.2% bytes.makeSlice
   113.5   5.3%  60.0%    115.4   5.4% code.google.com/p/go.net/html.(*Tokenizer).Token
    78.3   3.6%  63.6%     78.3   3.6% code.google.com/p/go.net/html.(*parser).addText
    68.4   3.2%  66.8%     68.4   3.2% strings.Map
 ...
    41.7   1.9%  77.2%     41.7   1.9% concatstring
***   37.7   1.7%  78.9%    736.6  34.2% main.ProcessResp
    27.9   1.3%  80.2%     27.9   1.3% makemap_c
 ...
    12.8   0.6%  91.8%     44.5   2.1% bitbucket.org/hooray-976/shuultuur/db.GraphBuild
    12.5   0.6%  92.4%     12.5   0.6% strings.genSplit
***   10.7   0.5%  92.9%    595.5  27.6% main.getContentFromHtml
 ...
```

*Fig. 2: Pprof result - Memory allocation in after imrovement*

The Figure 3 is a top report that shows CPU usage in the beginning of development and it was very high.

As you can see in Figure 4, the following top report shows much less CPU usage after optimizing the banned and weighted phrase search.

Figure 9 (see Appendix) is a gcvis graph shows memory usage when the program was not optimized.

Figure 10 (see Appendix) shows memory usage after some optimizations.

We have implemented a number of other improvements such as learning URL/Domains to not check banned and weighted phrases every time in HTTP response bodies. The learned mode feature was added something like in following way:

```
...
// Learn and store this URL to redisdb
// temporarily use xxhash to get
// checksum from URL/Domain
blob1 := []byte(requrl)
h32g := xxh.GoChecksum32(blob1)

// key = 0xXXXXXXXX for expire_time seconds,
// 1 for BLOCK, 2 for PASS
key := fmt.Sprintf("%s0x%08x", policy, h32g)

// SET key value [EX seconds]
// [PX milliseconds] [NX|XX]
db.Exec("SET", key, BLOCK, "EX", EXPIRE, "NX")
...
```

Another improvement we did was a possibility to limit the listener to accept a specified number of simultaneous connections. Rate limiting on incoming requests was done again utilizing Redis like:

```
...
// Will set this via config file
limit := 10

// Increment counter for request
// (this will create a new key
// if it does not exist)
current, err := redis.Int(db.Incr
              (url_path + remote_addr))
...

// Check if the returned counter
// exceeds our limit
if current > limit {
    fmt.Println(">>> Too many requests -",
           url_path + remote_addr)
    response := goproxy.NewResponse(request,
           goproxy.ContentTypeHtml, 429,
           "Too many requests!")
    return request, response
} else if current == 1 {
    fmt.Println(">>> SET counter for:",
           url_path + remote_addr)

    // Set expiry on fresh counter for the
    // given url_path and remote address
    db.Exec("SETEX", url_path +
                    remote_addr, 1, 1)
}
 ...
```

- Slow image filtering on HTTP response. It is temporarily disabled until we find a proper solution.
- One last major issue could be related to the high number of goroutines under heavy load, which results in high CPU and memory usage. Currently we are investigating the issue [17].

## 5 Benchmark results

**Case 1:**

In order to compare performance of our implementation in contrast to existing solutions, we have used Dansguardian-2.12.0.3 and tested it in the same environment. We know that Dansguardian is usually used with squid, therefore we used squid version 3.4.8_2 in our test. At last, our content filtering software, Shuultuur, was written in Go 1.3.2 on FreeBSD/amd64. We used the same server for this performance test comparison and the Internet link speed was

```
…
lastpid:  1189;  load averages:  7.30,  2.42,  0.93 up 0+00:30:51  14:57:41
61 processes:  1 running, 60 sleeping
CPU: 20.5% user,  0.0% nice, 42.0% system,  6.6% interrupt, 31.0% idle
Mem: 104M Active, 63M Inact, 225M Wired, 234M Buf, 7502M Free
Swap: 16G Total, 16G Free

  PID USERNAME     THR PRI NICE   SIZE    RES STATE   C   TIME   WCPU COMMAND
 1131 tsgan         22  52    0   182M 46196K uwait   4   9:29 685.50% shuultuur
  900 redis          3  52    0 69952K 42512K uwait   6   1:11 88.48% redis-server
 1130 tsgan          6  20    0 37856K  9084K piperd  1   0:01  0.00% gcvis
  918 tsgan          1  20    0 72136K  5832K select  5   0:00  0.00% sshd
  889 squid          1  20    0 70952K 16412K kqread  5   0:00  0.00% squid
 1049 tsgan          1  20    0 38388K  5168K select 11   0:00  0.00% ssh
  998 tsgan          1  20    0 72136K  5904K select  9   0:00  0.00% sshd
  919 tsgan          1  20    0 17564K  3528K pause   2   0:00  0.00% csh
  868 root           1  20    0 22256K  3284K select 11   0:00  0.00% ntpd
…
```

*Fig. 3: Top report - In initial stage*

```
…
lastpid:  1253;  load averages:  0.15,  0.31,  0.32 up 0+00:55:22  11:55:42
45 processes:  1 running, 44 sleeping
CPU:  1.4% user,  0.0% nice,  0.0% system,  0.0% interrupt, 98.6% idle
Mem: 96M Active, 72M Inact, 279M Wired, 310M Buf, 7445M Free
Swap: 16G Total, 16G Free

  PID USERNAME     THR PRI NICE   SIZE    RES STATE   C   TIME   WCPU COMMAND
 1183 root          17  20    0   142M 37348K uwait   0   7:28 14.31% shuultuur
  896 redis          3  52    0 78144K 62896K uwait   3   0:52  0.00% redis-server
 1182 root           6  20    0 45048K 16840K uwait   9   0:16  0.00% gcvis
  993 tsgan          1  20    0 72136K  6744K select  9   0:06  0.00% sshd
 1187 tsgan          1  20    0  9948K  1600K kqread 10   0:03  0.00% tail
 1091 tsgan          1  20    0 16596K  2548K CPU8    8   0:02  0.00% top
 1204 tsgan          1  20    0 38388K  5164K select  5   0:00  0.00% ssh
 1196 tsgan          1  20    0 72136K  5904K select  1   0:00  0.00% sshd
  885 squid          1  20    0 70952K 16384K kqread  0   0:00  0.00% squid
…
```

*Fig. 4: Top report - After improvement*

5Mbps. The server's technical specification is listed below:

- CPU - Intel(R) Xeon(R) X5670 2.93GHz
- Memory - 8192MB
- FreeBSD/SMP -12 CPUs (package(s) x 6 core(s) x 2 SMT threads)

We used FreeBSD 9.2-RELEASE and /etc/sysctl.conf includes following:

- kern.ipc.somaxconn = 27737
- kern.maxfiles = 123280
- kern.maxfilesperproc = 110950
- kern.ipc.maxsockets = 85600
- kern.ipc.nmbclusters = 262144
- net.inet.tcp.maxtcptw = 47120

We also had to change tcp-backlog setting to high value in the Redis config file. Furthermore, we performed HTTP load test using http_load-14aug2014 (parallel and rate test) [18] for both Dansguardian and Shuultuur. In http_load test, we used following URLs:

- http://fxr.watson.org/fxr/source /arm/lpc/lpc_dmac.c
- http://www.news.mn/news.shtml
- http://mongolian-it.blogspot.com/
- http://www.patrick-wied.at/static/nudejs/demo/
- http://news.gogo.mn/
- http://www.amazon.com/
- http://edition.cnn.com/?refresh=1
- http://www.uefa.com/
- http://www.tmall.com/
- http://www.reddit.com/r/aww.json
- http://nginx.com
- http://www.yahoo.com
- http://slashdot.org/?nobeta=1
- http://www.ikon.mn
- http://www.gutenberg.org
- http://en.wikipedia.org/ wiki/BDSM
- http://www3.nd.edu/dpettifo /tutorials/testBAD.html
- http://penthouse.com/# cover_new?{}
- http://www.playboy.com

| No | Result names | | Parallel test | | Rate test | |
|---|---|---|---|---|---|---|
| | | | Shuultuur | Dansguardian | Shuultuur | Dansguardian |
| 1 | Fetches | | 17654 | 4298 | 5991 | 5389 |
| 2 | Max parallel | | 10 | 10 | 95 | 606 |
| 3 | Mean bytes/connection | | 79213.8 | 94820.7 | 72666.3 | 27437.2 |
| 4 | Fetches/sec | | 29.4233 | 7.16333 | 9.985 | 8.98166 |
| 5 | Msecs/connect | | 0.189717 mean, 13.855 max, 0.088 min | 0.184428 mean, 0.485 max, 0.088 min | 0.177924 mean, 2.037 max, 0.106 min | 0.345489 mean, 0.782 max, 0.12 min |
| 6 | Msecs/first-response | | 229.182 mean, 5114.55 max, 8.049 min | 1374.9 mean, 40977.9 max, 0.779 min | 1189.41 mean, 59271.7 max, 11.144 min | 26442.1 mean, 59925.3 max, 3.322 min |
| 7 | Timeouts | | - | - | 107 | 3432 |
| 8 | Bad byte counts | | 6660 | 1415 | 2470 | 3691 |
| 9 | HTTP response codes | 200 | 12120 | 3595 | 4015 | 1744 |
| 10 | | 301 | 714 | 191 | 249 | 105 |
| 11 | | 302 | 819 | 171 | 273 | 114 |
| 12 | | 403 | 3843 | - | 1325 | - |
| 13 | | 404 | 10 | - | - | - |
| 14 | | 500 | 148 | - | 70 | - |
| 15 | | 503 | - | 341 | - | - |

*Fig. 5: Performance test result (Server)*

- http://www.bbc.com/earth /story/20141020-chicks-tumble-of-terror-filmed
- http://173.244.215.173/go /indexb.html
- http://breakingtoonsluts. tumblr.com/

Some of above URLs are listed in the Shallalist blacklist, some URLs contain phrases which are in the banned and weighted phrase lists, some URLs have lots of content and javascript and rest of the URLs are chosen with no particular reason. The following test commands used for HTTP load tests:

```
./http_load -proxy 172.16.2.1:8080 -parallel 10
                          -seconds 600 urls
./http_load -proxy 172.16.2.1:8080 -rate 10
                         -jitter -seconds 600 urls
```

The option *-parallel* in the first command indicates the number of concurrent connections to establish and maintain, the *-rate* option in the second command controls number of requests sent out per second, the *-jitter* option varies the rate by about 10%, and the *-seconds* option indicates the number of seconds to run the test. Figure 5 shows the comparison table of http_load parallel and rate test results.

Based on the above result Shuultuur has some advantages and disadvantages. For example, since Shuultuur is still under development, it responded with Internal Server Error (500) more often than Dansguardian. On the other hand, Shuultuur responded with much more successful responses (200). Dansguardian has some limitations and it responded 341 times with Service Unavailable (503) and had much more timeouts. On the performance side, in average, Shuultuur's performance was higher than Dansguardian in most cases for both tests.

**Case 2:**

In this case, the scenario is almost the same as in Case 1, but we used different hardware (APU system board) [20], updated Go to 1.4.1 and changed the Internet link speed to 2Mpbs. The hardware's technical specification is listed below:

- CPU -AMD G series T40E, 1 GHz dual Bobcat core with 64 bit support, 32K data + 32K instruction + 512K L2 cache per core
- Memory - 4096MB

On APU, we used FreeBSD 10.1-RELEASE and /etc/sysctl.conf includes following:

- kern.ipc.somaxconn = 4096
- kern.maxfiles = 10000
- kern.maxfilesperproc = 8500
- kern.ipc.maxsockets = 6500
- kern.ipc.nmbclusters = 20000
- net.inet.tcp.maxtcptw = 4000

Because of the smaller hardware we had to change tcp-backlog setting to 4096 in the Redis config file. In this case, we also used HTTP load test using http_load-03feb2015 (parallel and rate test) [18] for both Dansguardian and Shuultuur. Figure 6 shows the comparison table of http_load parallel and rate test results.

The Figure 11 and Figure 12 (see Appendix) shows memory usage on http_load test on

| No | Result names | | Parallel test | | Rate test | |
|---|---|---|---|---|---|---|
| | | | Shuultuur | Dansguardian | Shuultuur | Dansguardian |
| 1 | Fetches | | 4319 | 2643 | 5877 | 5225 |
| 2 | Max parallel | | 10 | 10 | 392 | 584 |
| 3 | Mean bytes/connection | | 120364 | 134945 | 103568 | 11322.7 |
| 4 | Fetches/sec | | 7.19813 | 4.405 | 9.795 | 8.70832 |
| 5 | Msecs/connect | | 19.193 mean, 3009.89 max, 0.925 min | 6.23727 mean, 53.385 max, 0.991 min | 13.3234 mean, 295.472 max, 0.721 min | 12.1561 mean, 3023.61 max, 0.903 min |
| 6 | Msecs/first-response | | 764.861 mean, 59830.3 max, 36.664 min | 1337.36 mean, 55849.5 max, 16.704 min | 8371.04 mean, 59971.6 max, 36.453 min | 35975.6 mean, 59984 max, 56.747 min |
| 7 | Timeouts | | 28 | 35 | 329 | 4618 |
| 8 | Bad byte counts | | 1787 | 2160 | 3023 | 4255 |
| 9 | HTTP response codes | 200 | 3677 | 2397 | 4181 | 542 |
| 10 | | 301 | 9 | 191 | 609 | - |
| 11 | | 302 | 366 | 217 | 458 | 70 |
| 12 | | 403 | 233 | - | 279 | - |
| 13 | | 404 | - | - | - | - |
| 14 | | 500 | 5 | - | 38 | - |
| 15 | | 503 | - | - | - | - |

*Fig. 6: Performance test result (APU)*

Shuultuur during rate and parallel tests respectively.

The above test results are similar to what we have observed during the tests that were done on the server. As in previous tests, Shuultuur's performance was higher than Dansguardian in most cases for both tests.

During the test time we captured top reports for both Shuultuur and Dansguardian, which are shown below figure 7 and 8.

As you can see from the above, the system load average especially CPU usage was high when Shuultuur was working.

## 6 Conclusions and Future Work

Developing application in Go using its useful built-in data structures such as maps and slices were simple and mostly straight forward. We were able to make a first working prototype in a matter of days. There were many open source projects written in Go in online source code repositories such as GitHub and many of those projects were very helpful for our development. The test results in Section 5 are results of only two cases. So far, we made http_load test multiple times and results were consistent. We expect that when we reach at first stable version the result will be lot better. As mentioned before, our implementation lacks fast and stable image checking feature. In the future work, we will improve image checking and we have to solve high number of goroutines problem described in Section 4. Finally, the memory usage and CPU load problem is a major issue for embedded system applications and we are planning to do more research on this to stabilize the resource usages.

## Acknowledgments

We would like to thank Christoph Badura from NetBSD project for his helpful comments and suggestions on this document.

```
last pid:  1317;  load averages:  1.52,  1.00,  0.58
71 processes:  1 running, 64 sleeping, 6 stopped
CPU: 31.4% user,  0.0% nice,  5.9% system,  1.6% interrupt, 61.2% idle
Mem: 58M Active, 189M Inact, 158M Wired, 70M Buf, 3519M Free
Swap: 978M Total, 978M Free

  PID USERNAME    THR PRI NICE   SIZE    RES STATE   C   TIME    WCPU COMMAND
 1300 user         18  25    0 84540K 43672K uwait   1   6:16  91.85% shuultuur
 1299 user          5  21    0 28544K  9484K piperd  1   0:18   4.10% gcvis
  822 redis         3  52    0 28108K  6540K uwait   1   0:21   0.29% redis-server
 1024 root          1  20    0 43580K 17092K select  0   3:42   0.00% dansguardian
  794 squid         1  20    0   164M 68400K kqread  1   1:20   0.00% squid
 1030 nobody        1  20    0 43580K 18660K select  1   0:02   0.00% dansguardian
 1028 nobody        1  20    0 43580K 18664K select  1   0:02   0.00% dansguardian
```

*Fig. 7: Top report for Shuultuur*

```
last pid:  1151;  load averages:  0.42,  0.68,  0.81
156 processes: 1 running, 152 sleeping, 3 stopped
CPU:  0.2% user,  0.0% nice, 10.2% system,  1.8% interrupt, 87.8% idle
Mem: 103M Active, 245M Inact, 161M Wired, 58M Buf, 3415M Free
Swap: 978M Total, 978M Free

  PID USERNAME    THR PRI NICE   SIZE    RES STATE   C   TIME    WCPU COMMAND
 1024 root          1  35    0 43580K 17092K nanslp  0   1:13  23.49% dansguardian
  794 squid         1  26    0   160M 62060K kqread  0   0:13   4.59% squid
 1002 user         19  42    0 93636K 51320K STOP    0   9:58   0.00% shuultuur
 1001 user          6  20    0 33856K 10692K STOP    0   0:32   0.00% gcvis
  822 redis         3  52    0 28108K  6452K uwait   1   0:15   0.00% redis-server
  932 user          1  20    0 21916K  3244K CPU0    0   0:06   0.00% top
 1028 nobody        1  20    0 43580K 18152K select  0   0:01   0.00% dansguardian
 1033 nobody        1  20    0 43580K 18172K select  0   0:01   0.00% dansguardian
  926 user          1  20    0 86472K  7240K select  1   0:01   0.00% sshd
 1025 nobody        1  20    0 31292K  5328K select  1   0:00   0.00% dansguardian
 1030 nobody        1  20    0 43580K 18304K select  0   0:00   0.00% dansguardian
 1053 nobody        1  20    0 43580K 18664K select  0   0:00   0.00% dansguardian
 1051 nobody        1  20    0 43580K 18180K select  1   0:00   0.00% dansguardian
 1059 nobody        1  20    0 43580K 18252K select  1   0:00   0.00% dansguardian
 1064 nobody        1  20    0 43580K 18244K select  0   0:00   0.00% dansguardian
 1029 nobody        1  20    0 43580K 18164K select  1   0:00   0.00% dansguardian
 1027 nobody        1  20    0 43580K 18624K select  0   0:00   0.00% dansguardian
  917 user          1  20    0 86472K  7192K select  0   0:00   0.00% sshd
 1034 nobody        1  20    0 43580K 18192K select  1   0:00   0.00% dansguardian
 1037 nobody        1  20    0 43580K 18208K select  0   0:00   0.00% dansguardian
 1026 nobody        1  20    0 31292K  5272K select  1   0:00   0.00% dansguardian
```

*Fig. 8: Top report for Dansguardian*

## References


[1] Half a decade with Go Retrieved from The Go blog: http://blog.golang.org/5years

[2] Retrieved from Go language resources: http://golang.cat-v.org/pure-go-libs

[3] Go (programming language). Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Go_%28programming_language%29

[4] The Go programming language. Retrieved from http://golang.org/

[5] Computer Language Benchmarks Game. Retrieved from http://benchmarksgame.alioth.debian.org/

[6] String Matching Algorithms and their Applicability in various Applications. (D. G. Nimisha Singla, Ed.) International Journal of Soft Computing and Engineering (IJSCE), 1 (6).

[7] Fast Cache for Your Text: Accelerating Exact Pattern Matching with Feed-Forward Bloom Filters. School of Computer Science. Pittsburgh, USA: Carnegie Mellon University.

[8] FreeBSD. Retrieved from https://www.freebsd.org/internet.html

[9] goproxy. (E. Leibovich) Retrieved from https://github.com/elazarl/goproxy

[10] gcvis. (D. Cheney) Retrieved from https://github.com/davecheney/gcvis

[11] profile. (D. Cheney) Retrieved from https://github.com/davecheney/profile

[12] go-nude. (Koyachi) Retrieved from https://github.com/koyachi/go-nude

[13] xxxhash-go. (S. Bunel) Retrieved from https://bitbucket.org/StephaneBunel/xxhash-go

[14] powerwalk. (Stretchr) Retrieved from https://github.com/stretchr/powerwalk

[15] redigo. (G. Burd) Retrieved from https://github.com/garyburd/redigo

[16] Redis. Retrieved from http://redis.io

[17] HTTP ListenAndServe Goroutines throughput. Retrieved from http://grokbase.com/t/gg/golang-nuts/147b9nb2nq/go-nuts-http-listenandserve-goroutines-throughput

[18] HTTP Load. (ACME Lab) Retrieved from http://acme.com/software/http_load/

[19] Profiling Go programs. Retrieved from https://blog.golang.org/profiling-go-programs

[20] PC engine APU board. http://www.pcengines.ch/apu1d4.htm

[21] Goroutines. Retrieved from https://golang.org/doc/effective_go.html#goroutines

[22] Channels. Retrieved from https://golang.org/doc/effective_go.html#channels

[23] Go maps in action. Retrieved from https://blog.golang.org/go-maps-in-action

[24] Arrays, slices (and strings): The mechanics of 'append'. Retrieved from https://blog.golang.org/slices

## APPENDIX A

./shuultuur

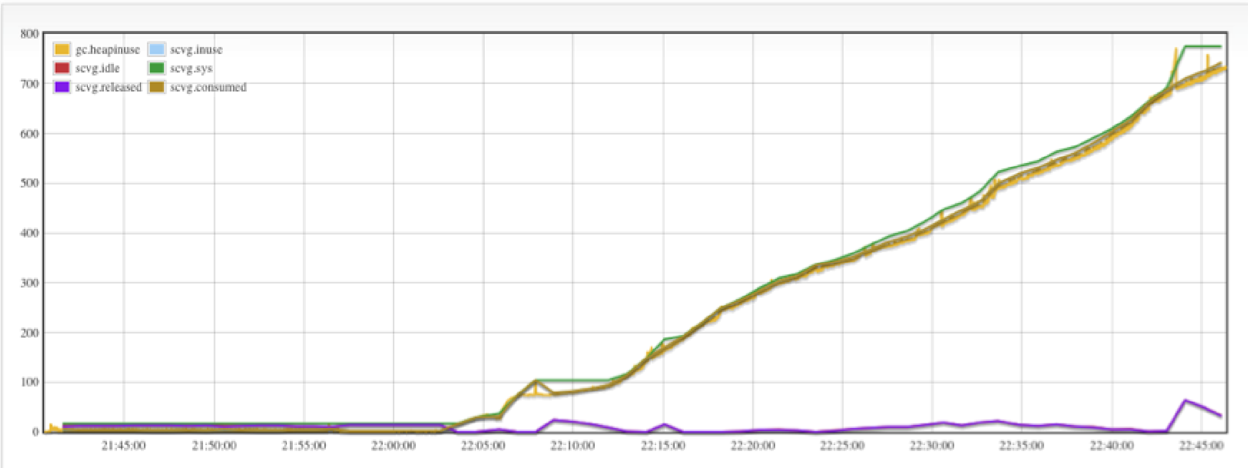


*Fig. 9: Memory usage before optimization*

./shuultuur

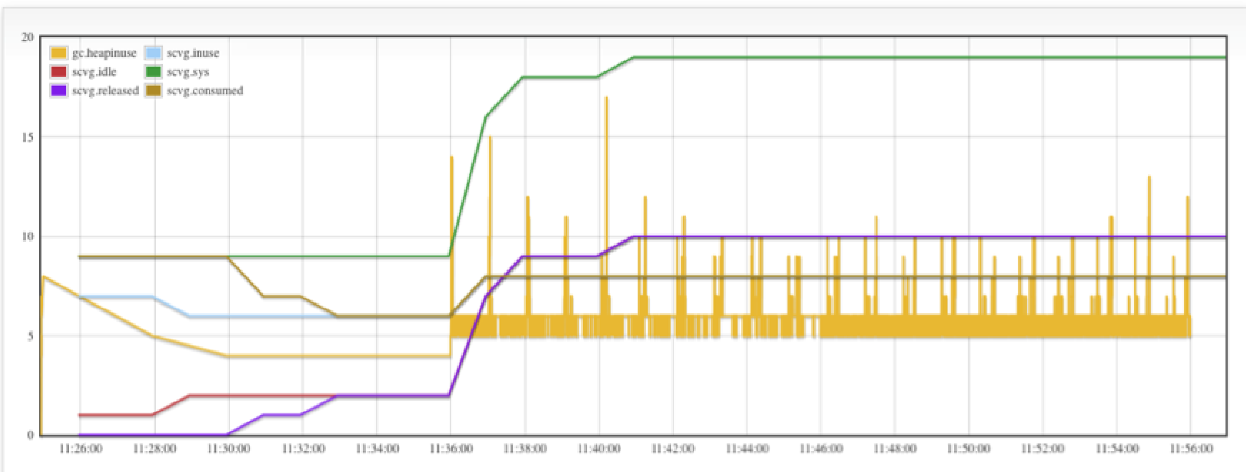


*Fig. 10: Memory usage after optimization*


*Fig. 11: Rate test*

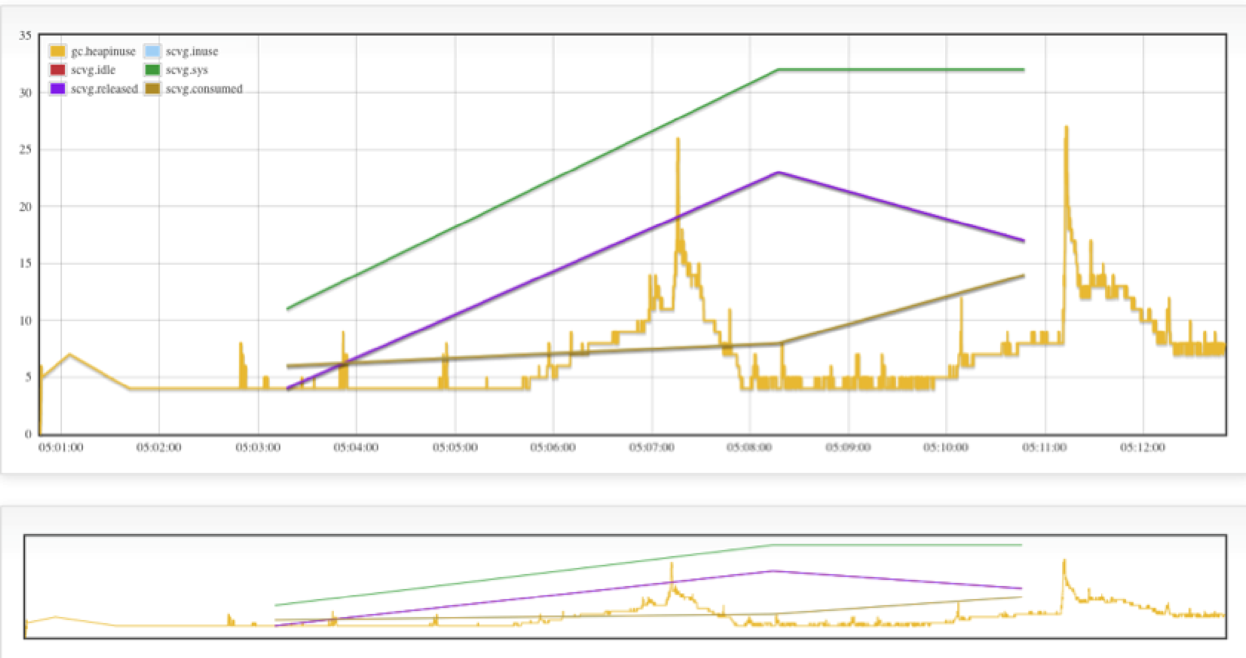


*Fig. 12: Parallel test*