Go based content filtering software on FreeBSD

Ganbold Tsagaankhuu, Mongolian Unix User Group
Esbold Unurkhaan, Mongolian University of Science and Technology
Erdenebat Gantumur, Mongolian Unix User Group

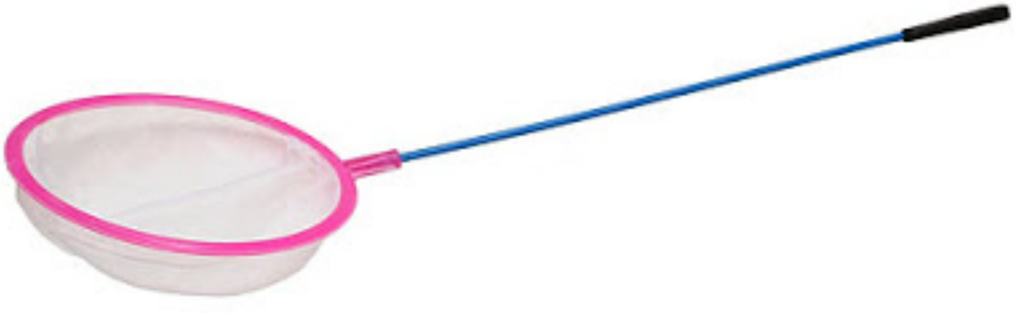
AsiaBSDCon
Tokyo, 2015

Content

- Introduction
- Rationale behind our choices
- Related projects
- Experienced challenges
- Benchmark Case 1, 2 and results
- Conclusions and future works

Introduction

- What is the meaning of Shuultuur ?



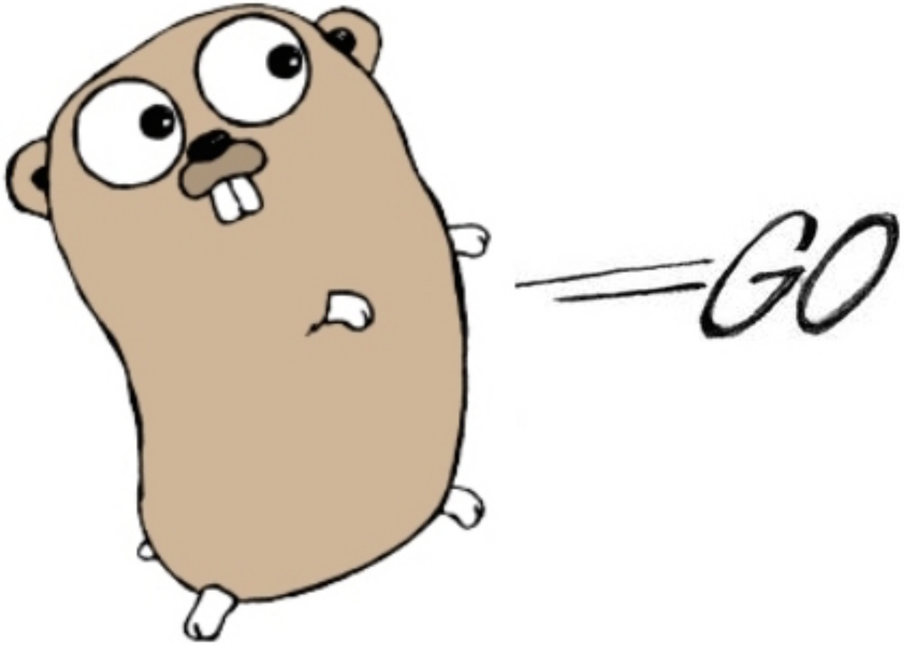
Шүүлтүүр

Rationale behind our choices

- Why content filter?
  - Some control over unwanted content from web
    - Enforce security policies in corporates
    - Parental control
    - Schools
    - Libraries
    - Inappropriate content depending from age
      - Adult
      - Violence
      - Drugs etc.

Rationale behind our choices

- Why Go?
  - Fast, lightweight, easy to prototype
  - Productive
  - Performance



Rationale behind our choices

- Why Go?
  - Go is
    - Compiled, statically typed
    - Garbage collected
    - Object oriented
  - Performance of Go’s
    - Somewhat comparable to C
    - Better than some of interpreted languages
  - Concurrency
    - Part of the programming language features
    - It has strong support for multiprocessing

Rationale behind our choices

- Why Go?
  - Go includes multiple useful built-in data structures such as maps and slices
  - Goroutines 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
    - In contrast coroutines communicate via yield and resume operations
  - Built-in profiling tool
  - Extensive number of libraries
  - BSD licensed

Rationale behind our choices

- Why FreeBSD is platform of choice?
  - Powerful, mature and stable
  - Complete, reliable and self-consistent distribution
  - FreeBSD's networking stack is very solid and fast
  - Easy to install and deploy the necessary applications and software using port and package system
  - Making custom FreeBSD image easily (such as NanoBSD)
  - We love FreeBSD

Related projects

- *goproxy*
  - Customizable HTTP proxy library for Go.
    - Supports regular HTTP proxy,
    - HTTPS through CONNECT,
    - "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
- *gcvis*
  - Visualizes Go program gctrace data in real time
- *profile*
  - Simple profiling support package for Go
- *go-nude*
  - Nudity detection with Go

Related projects

- *xxhash-go*
  - Go wrapper for C xxhash - an extremely fast Hash algorithm
  - Working at speeds close to RAM limits
- *powerwalk*
  - Go package for walking files
  - Concurrently calling user code to handle each file
- *redigo*
  - Go client for the Redis database
- *Redis*
  - Open source, BSD licensed, advanced *key-value cache* and *store*

Experienced challenges

- Problems during development:
  - The Shallalist blacklist
    - 1.8 million URL/Domain entries.

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

Experienced challenges

- Solution. Changed the code to:

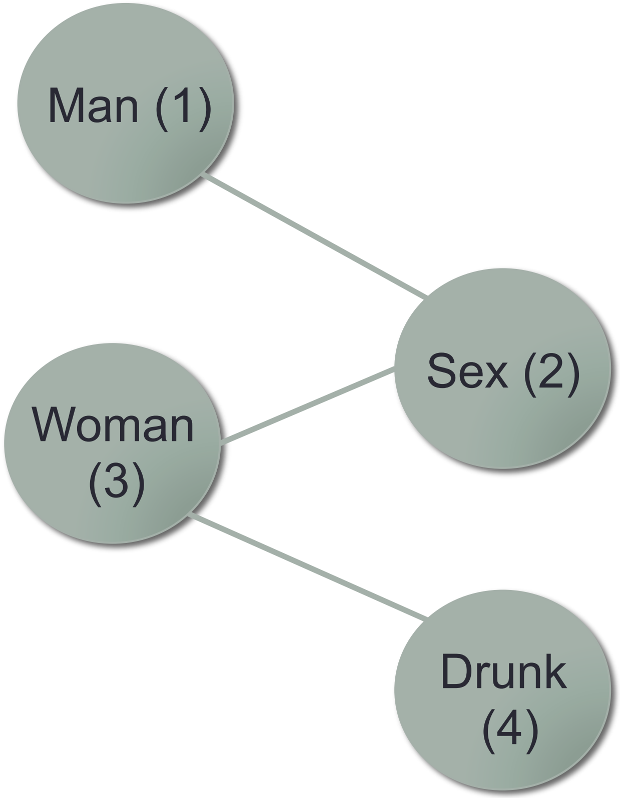
```
…
// use xxhash to get checksum from URL/Domain
blob := []byte(url_or_domain)
h32g := xxh.GoChecksum32(blob)

/*
 * 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)
…
```

Experienced challenges

- Banned and weighted phrase lookup problem
    - Problem: Storing all phrases in Redis
        - Slow and not efficient
        - Loop is expensive
    - Solution: Graph and map
        - Every unique word is an edge of the graph
        - Edges and Vertices are stored in the map
            - Map – Go’s implementation of hash table
    - Problem: Regular expression based search
        - CPU intensive
    - Solution: Graph and Boyer Moore search algorithm

Experienced challenges

Graph representation



For example: “sex woman”, “sex man” and *“drunk woman sex” words in Graph.*

Man: 2-1

Sex: 2-1, 2-3, 4-3-2

Drunk: 4-3-2

Woman: 2-3, 4-3-2

Experienced challenges

- Reading HTTP response bodies into memory
  - Heap memory usage grow very large
    - Lots of allocations
    - When the rate of connections per second is high
- Solution
  - Streaming parser by utilizing the io.Reader interface
  - Limiting incoming requests
  - CPU and memory profiling
    - Go’s built-in profiler pprof

Experienced challenges

```
# 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)
```

Experienced challenges

```
# 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
   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
…
    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
…
```

Experienced challenges

- CPU usage

```
…
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
…
```

Experienced challenges

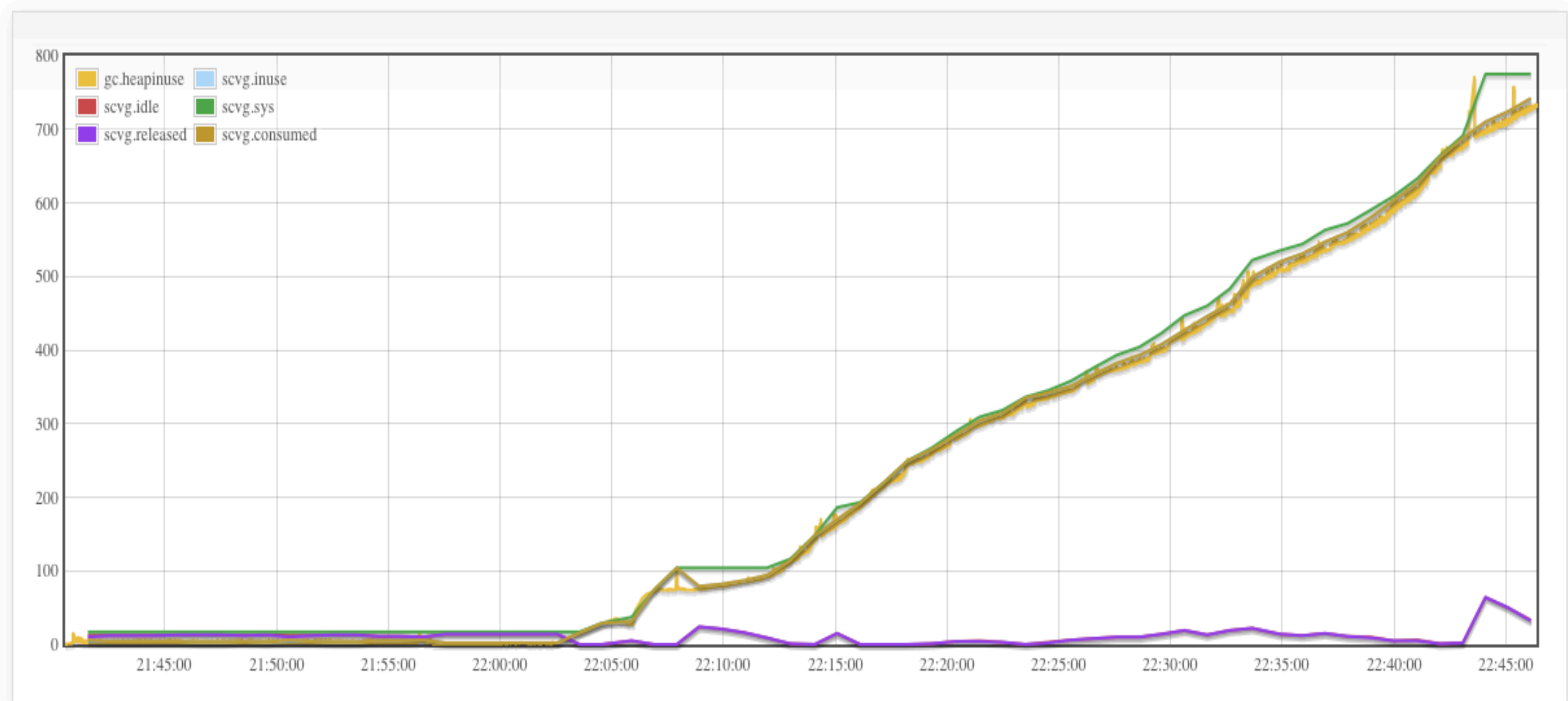
- CPU usage after optimizations

```
…
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
…
```

Experienced challenges

- Memory usage

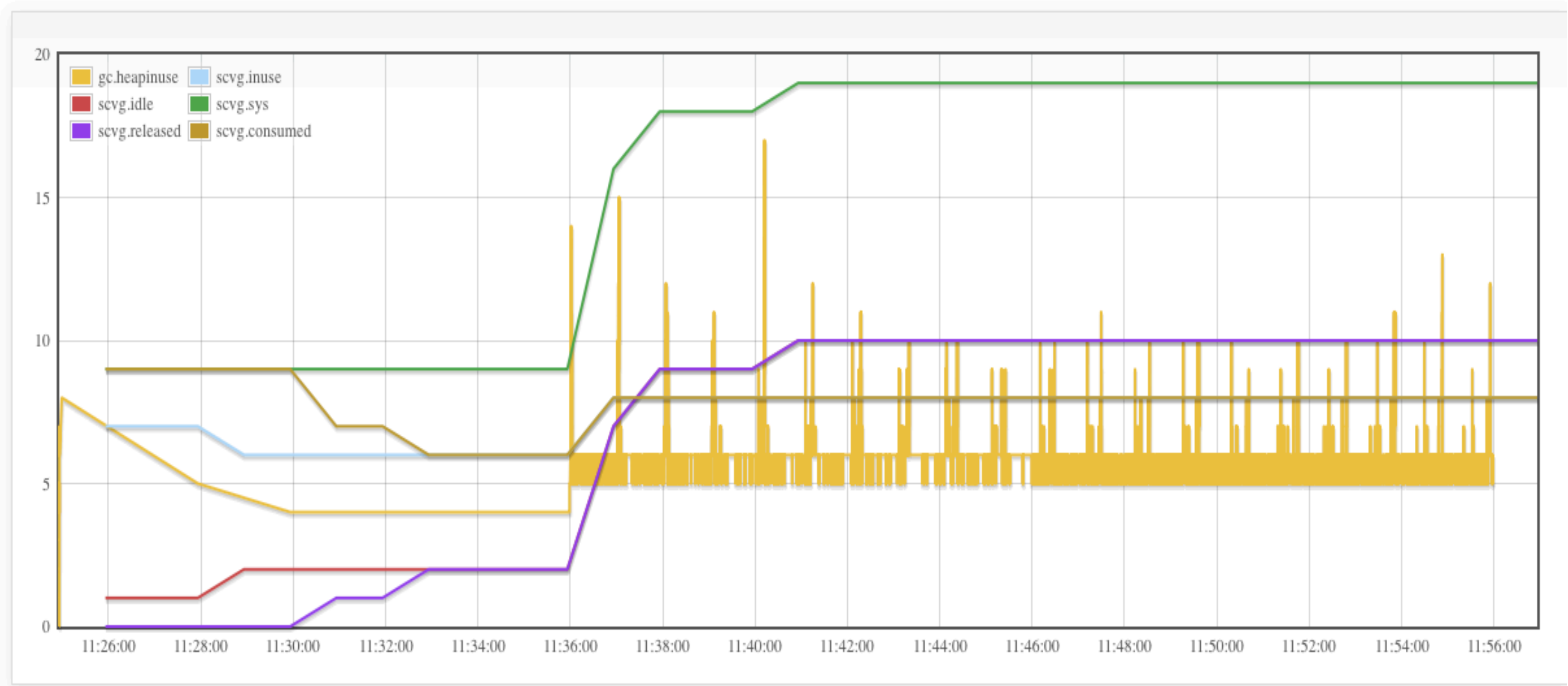
./shuultuur



Experienced challenges

- Memory usage after optimizations

./shuultuur



Experienced challenges

- Other improvements
  - Learned mode (caching)
    - To not check HTTP response bodies every time
  - Rate limiting on incoming requests utilizing Redis
  - Limit the listener to accept a specified number of simultaneous connections

Experienced challenges

- Learned mode

```
…
// 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")
…
```

Experienced challenges

- Limit listener:

```
…
type Server struct {
        *http.Server
        ListenLimit int  // Limit the number of outstanding requests
}
func (srv *Server) ListenAndServe() error {
…
        l, err := net.Listen("tcp", addr)
        l = netutil.LimitListener(l, srv.ListenLimit)
        return srv.Serve(l)
}
…
if LISTEN_LIMIT_ENABLE == 1 {
        srv := &Server {
              ListenLimit: LISTEN_LIMIT,
              Server: &http.Server{Addr: ":8080", Handler: proxy},}
        log.Fatal(srv.ListenAndServe())
} else {
        log.Fatal(http.ListenAndServe(":8080", proxy))
}
```

Experienced challenges

- Slow image filtering on HTTP response
  - Used go-nude, but temporarily disabled until we find a proper solution
- High number of goroutines under heavy load
  - High CPU and memory usage.
  - Currently we are investigating the issue

Experienced challenges

- Problem: Our program panics sometimes with following message:
  - panic: dial tcp 127.0.0.1:6379: connection reset by peer
- Solution:
  - This was related to OS settings.
    - netstat -anL shows the limits.
    - Increased:
      - `kern.ipc.somaxconn sysctl value`
  - Increased tcp-backlog in redis.conf

Benchmark (Case 1)

- Test environment (Case 1):
  - Server OS
    - FreeBSD 9.2-RELEASE amd64
  - Server hardware:
    - CPU - Intel(R) Xeon(R) X5670 2.93GHz
    - Memory - 8192MB
    - FreeBSD/SMP -12 CPUs (package(s) x 6 core(s) x 2 SMT threads)
  - Go version 1.3.2
  - Dansguardian version 2.12.0.3
  - Squid version 3.4.8_2

Benchmark (Case 1)

- Increased some sysctl 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
```

Benchmark (Case 1)

- Increased tcp-backlog setting to high value in the Redis config file
- http_load-14aug2014 (parallel and rate test)
- Tested URL/Domains:
  - 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/

Benchmark (Case 1)

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

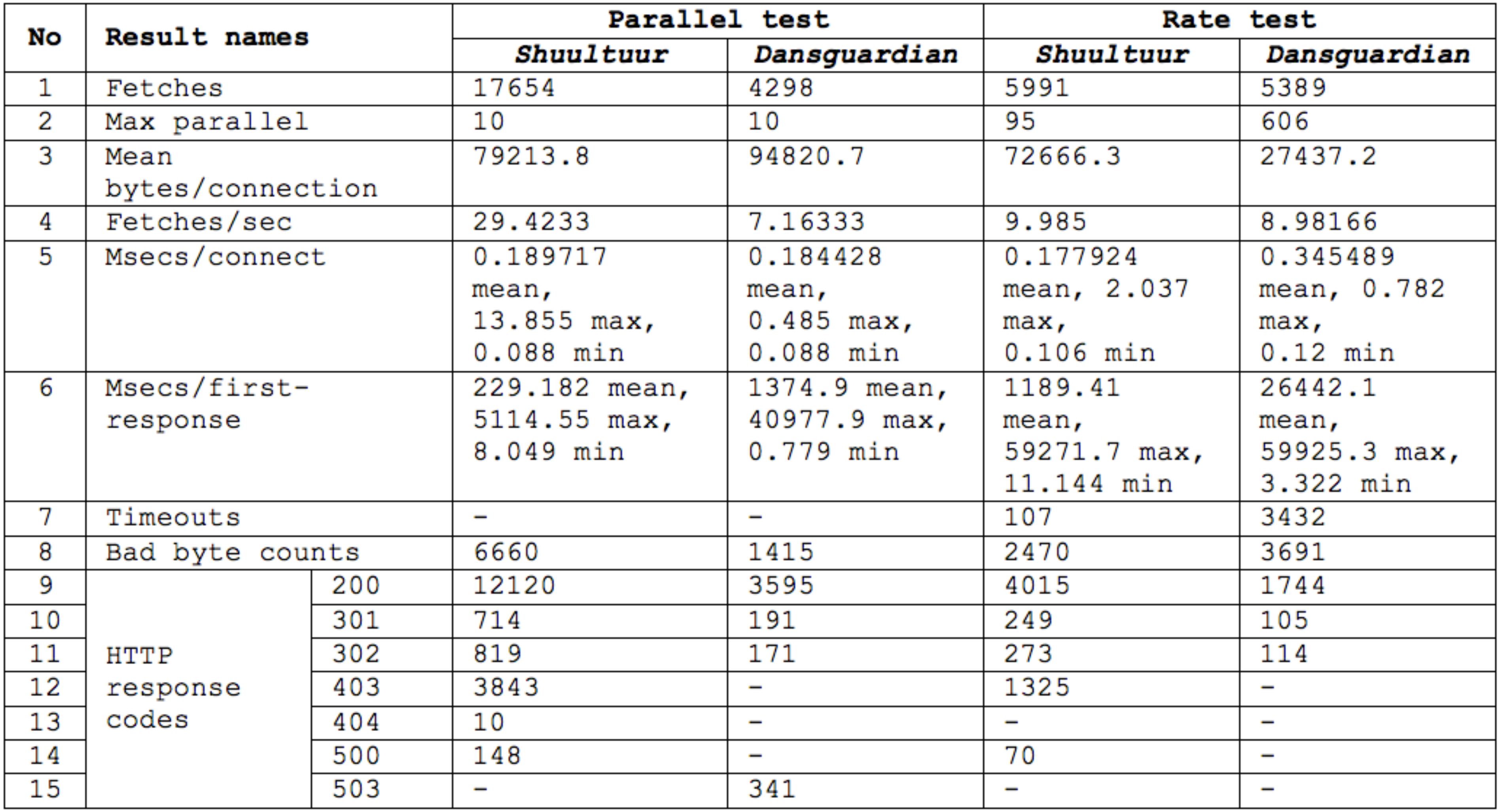
Benchmark (Case 1)

- 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
```

- -parallel : number of concurrent connections to establish and maintain
- -rate : number of requests sent out per second
- -jitter : varies the rate by about 10%
- -seconds : number of seconds to run the test

Benchmark (Case 1) results



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	-	-

Benchmark (Case 1) results

- Shuultuur has some advantages and disadvantages
  - Internal Server Error (500) more often than Dansguardian
  - More successful responses (200).
- Dansguardian
  - Responded 341 times with Service Unavailable (503)
  - Much more timeouts.
- On the performance side, in average, Shuultuur’s performance was higher than Dansguardian in most cases for both tests.

Benchmark (Case 2)

- Test environment (Case 2)
  - Server OS
    - FreeBSD 10.1-RELEASE amd64
  - Server hardware:
    - 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
  - Go version 1.4.1
  - Squid and Dansguardian versions are same as before

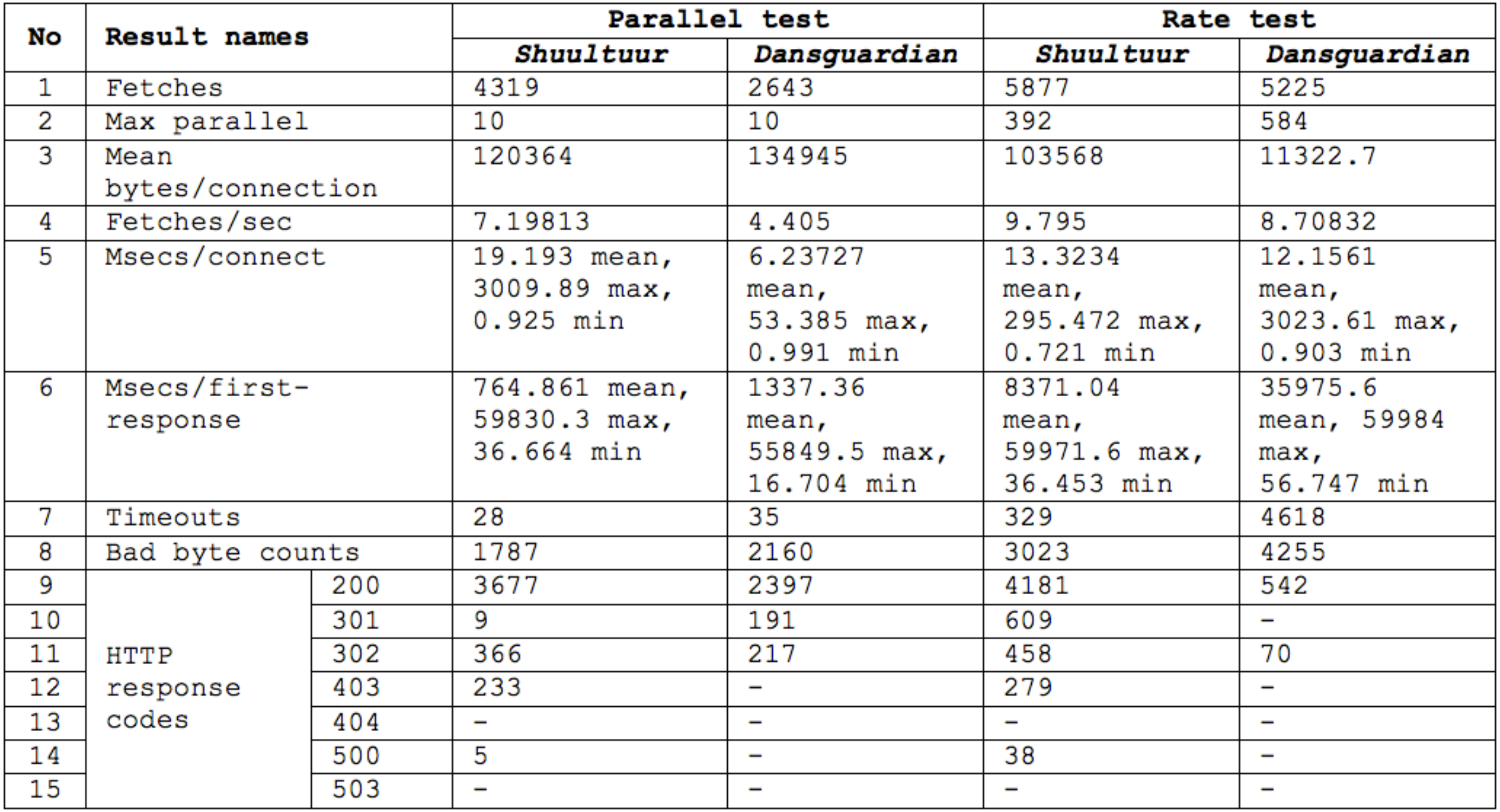
Benchmark (Case 2)

- /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
```

- Changed tcp-backlog setting to 4096 in the Redis config file
- http_load-03feb2015 (parallel and rate test)

Benchmark (Case 2) results



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	-	-	-	-

Benchmark (Case 2) results

- Shuultuur’s performance was higher than Dansguardian in most cases for both tests
- System load average especially CPU usage was high when Shuultuur was working

Benchmark (Case 2) results

- top report when running Shuultuur:

```
lastpid:  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
 1029 nobody        1  20    0 43580K 18672K select  1   0:02  0.00% dansguardian
 1033 nobody        1  20    0 43580K 18664K select  0   0:02  0.00% dansguardian
 1032 nobody        1  20    0 43580K 18660K select  0   0:02  0.00% dansguardian
 1031 nobody        1  20    0 43580K 18672K select  1   0:02  0.00% dansguardian
```

Benchmark (Case 2) results

- Dansguardian:

```
lastpid:  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
```

Conclusions and future works

- Developing application in Go is simple
  - Using built-in data structures such as maps and slices
  - Many open source projects were useful
- http_load test was run multiple times and results were consistent
- Results will be lot better when we solve problems

Conclusions and future works

- Lack of fast and stable image checking feature
- High number of goroutines problem when load is high
  - Use channels for incoming requests to have some queuing mechanism
- Last but not least
  - The memory usage and CPU load problem is a major issue for embedded system applications
  - Planning to do more research on this to stabilize the resource usages.
- Any comments and ideas related to Shuultuur
  - Contact: ganbold@gmail.com

Thank you for your attention

Questions?