H2O

the optimized HTTP server

DeNA Co., Ltd.

Kazuho Oku

Who am I?

- long experience in network-related / high-performance programming
- works in the field:
  - Palmscape / Xiino
    - world's first web browser for Palm OS, bundled by Sony, IBM, NTT DoCoMo
  - MySQL extensions: Q4M, mycached, …
    - MySQL Conference Community Awards (as DeNA)
  - JSX
    - altJS with an optimizing compiler

Agenda

- Introduction of H2O
- The motives behind
- Writing a fast server
- Writing H2O modules
- Current status & the future
- Questions regarding HTTP/2

Introducing H2O

H2O – the umbrella project

- h2o – the standalone HTTP server
  - libh2o – can be used as a library as well
- picohttpparser – the HTTP/1 parser
- picotest – TAP-compatible testing library
- qrintf – C preprocessor for optimizing s(n)printf
- yoml – DOM-like wrapper for libyaml

github.com/h2o

h2o

- the standalone HTTP server
- protocols:
  - HTTP/1.x
  - HTTP/2
    - via Upgrade, NPN, ALPN, direct
  - WebSocket (uses wslay)
  - with SSL support (uses OpenSSL)
- modules:
  - file (static files), reverse-proxy, reproxy, deflate
- configuration using yaml

libh2o

- h2o is also available as a library
- event loop can be selected
  - libuv
  - h2o's embedded event loop
- configurable via API and/or yaml
  - dependency to libyaml is optional

Modular design

- library layer:
  - memory, string, socket, timeout, event-loop, http1client, …
- protocol layer:
  - http1, http2, websocket, loopback
- handlers:
  - file, reverse-proxy
- output filters:
  - chunked-encoder, deflate, reproxy
- loggers:
  - access-log

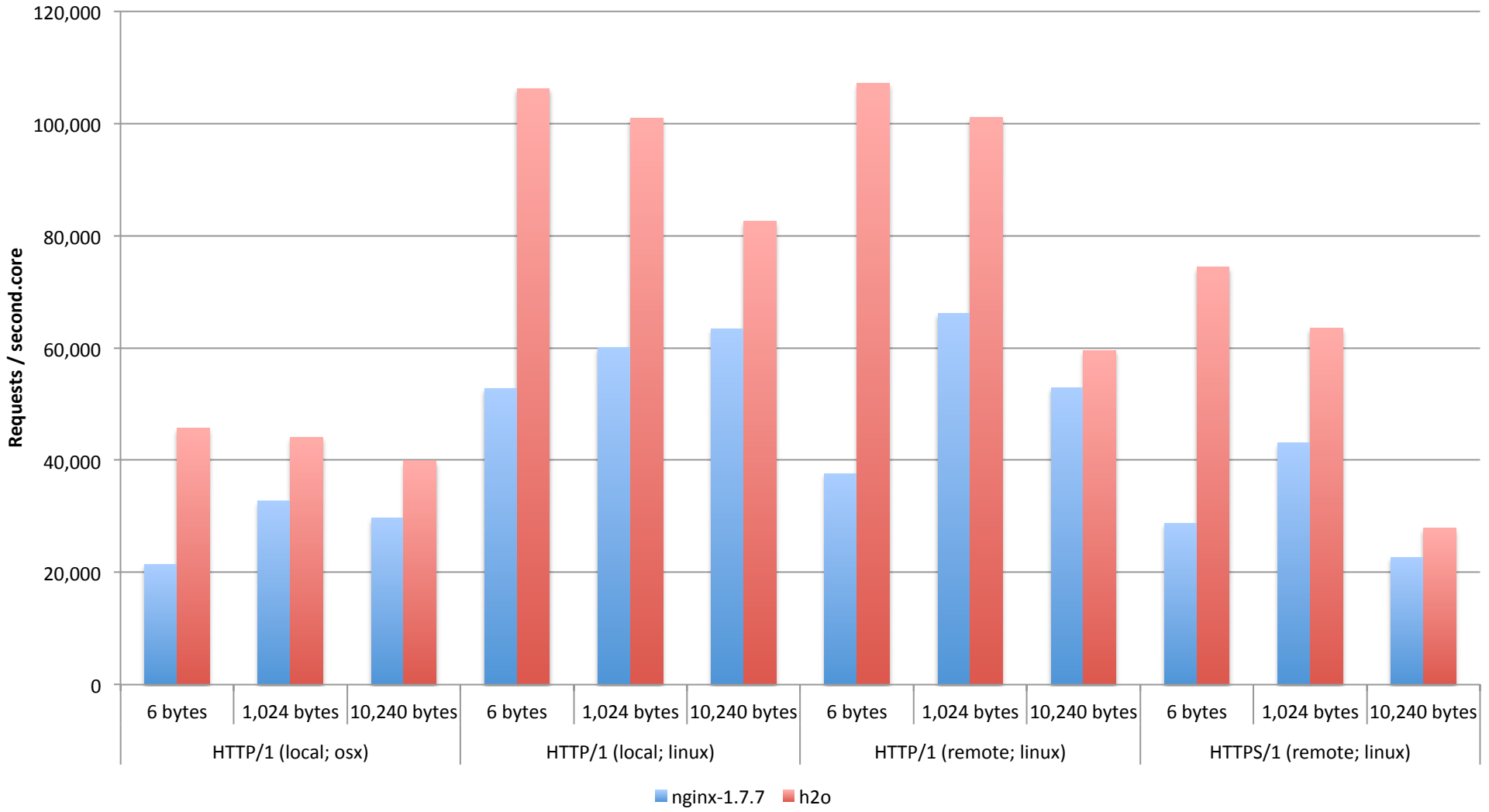
Testing

- two levels of testing for better quality
  - essential for keeping the protocol implementations and module-level API apart
- unit-testing
  - every module has (can have) it's own unit-test
  - tests run using the *loopback* protocol handler
    - module-level unit-tests do not depend on the protocol
- end-to-end testing
  - spawns the server and connect via network
  - uses nghttp2

Internals

- uses `h2o_buf_t` (pair of [char*, size_t]) is used to represent data
  - common header names are interned into tokens
    - those defined in HPACK static_table + α
- mostly zero-copy
- incoming data allocated using: malloc, realloc, mmap
  - requires 64-bit arch for heavy use
- uses writev for sending data

Fast



Note: used MacBook Pro Early 2014 (Core i7@2.4GHz), Amazon EC2 cc2.8xlarge, no logging

Why is it fast?

Why should it be fast?

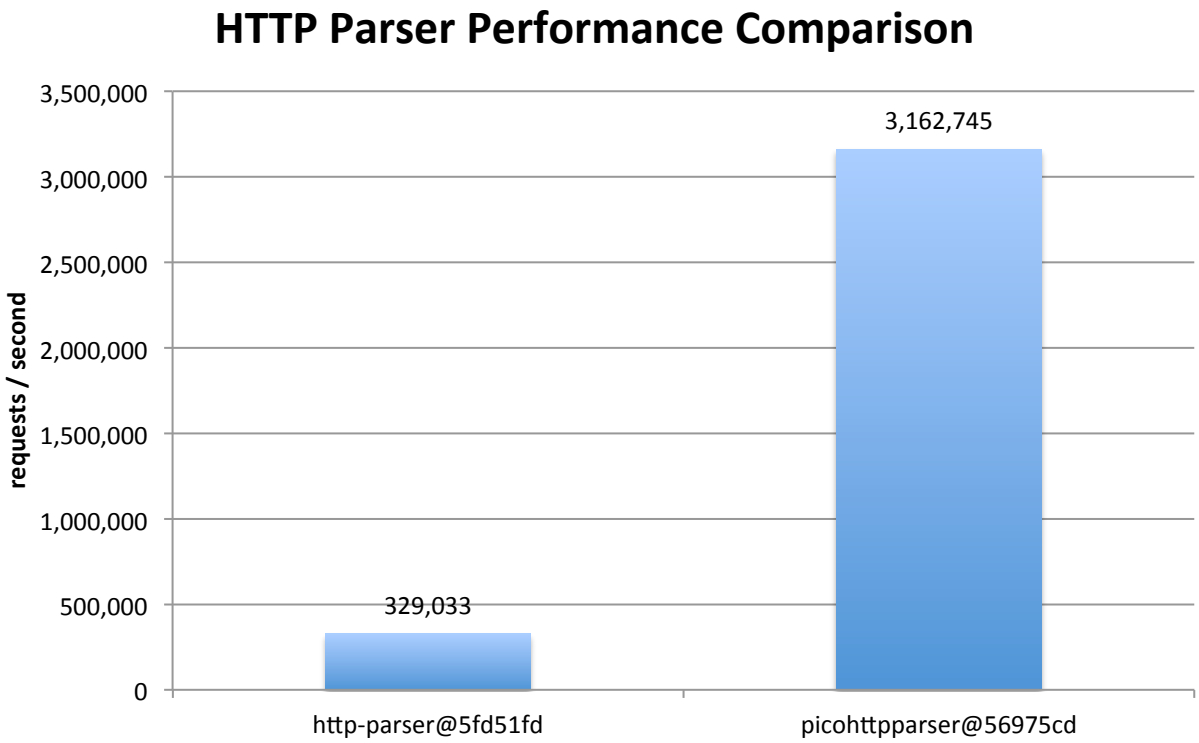
It all started with PSGI/Plack

- PSGI/Plack is the WSGI/Rack for Perl
- on Sep 7th 2010:
  - first commit to github.com/plack/Plack
  - I asked: why ever use FastCGI?
    - at the time, HTTP was *believed* to be slow, and FastCGI is necessary
      - the other choice was to use Apache+mod_perl
  - I proposed:
    - write a fast HTTP parser in C, and use it from Perl
    - get rid of specialized protocols / tightly-coupled legacy servers
      - for ease of dev., deploy., admin.

So I wrote HTTP::Parser::XS and picohttpparser.

How fast is picohttpparser?

- 10x faster than http-parser according to 3p bench.
  - github.com/fukamachi/fast-http



HTTP::Parser::XS

- the de-facto HTTP parser used by PSGI/Plack
  - PSGI/Plack is the WSGI/Rack for Perl
- modern Perl-based services **rarely use FastCGI or mod_perl**
- the application servers used (Starlet, Starman, etc.) speak HTTP using HTTP::Parser::XS
  - application servers can be and in fact are written in Perl, since the slow part is handled by HTTP::Parser::XS
- picohttpparser is the C-based backend of HTTP::Parser::XS

The lessons learned

- using one protocol (HTTP) everywhere reduces the TCO
  - easier to develop, debug, test, monitor, administer
  - popular protocols tend to be better designed & implemented thanks to the competition
- similar transition happens everywhere
  - WAP has been driven out by HTTP & HTML
  - we rarely use FTP these days

but HTTP is not yet used everywhere

- web browser
  - HTTP/1 is used now, transiting to HTTP/2
- SOA / microservices
  - HTTP/1 is used now
    - harder to transit to HTTP/2 since many proglangs use blocking I/O
  - other protocols coexist: RDBMS, memcached, …
    - are they the next target of HTTP (like FastCGI?)
- IoT
    - MQTT is emerging

So I decided to write H2O

- in July 2014
- life of the developers becomes easier if all the services use HTTP
- but for the purpose, it seems like we need to raise the bar (of performance)
  - or other protocols may emerge / continue to be used
- now (at the time of transition to HTTP/2) might be a good moment to start a performance race between HTTP implementers

Writing a fast server

Two things to be aware of

- characteristics of a fast program
  1. executes less instructions
     - speed is a result of simplicity, not complexity
  2. causes less pipeline hazards
     - minimum number of conditional branches / indirect calls
     - use branch-predictor-friendly logic
       - e.g. "conditional branch exists, but it is taken 95%"

H2O - design principles

- do it right
  - local bottlenecks can be fixed afterwards
  - large-scale design issues are hard to notice / fix
- do it simple
  - as explained
  - provide / use hooks only at high-level
    - hooks exist for: protocol, generator, filter, logger

The performance pitfalls

- many server implementations spend CPU cycles in the following areas:
  - memory allocation
  - parsing input
  - stringifying output and logs
  - timeout handling

Memory allocation

Memory allocation in H2O

- uses region-based memory management
  - "memory pool" of Apache
- strategy:
  - memory block is assigned to the Request object
  - small allocations returns portions of the block
  - memory is never returned to the block
  - The entire memory block gets freed when the Request object is destroyed

Memory allocation in H2O (cont'd)

- malloc (of small chunks)

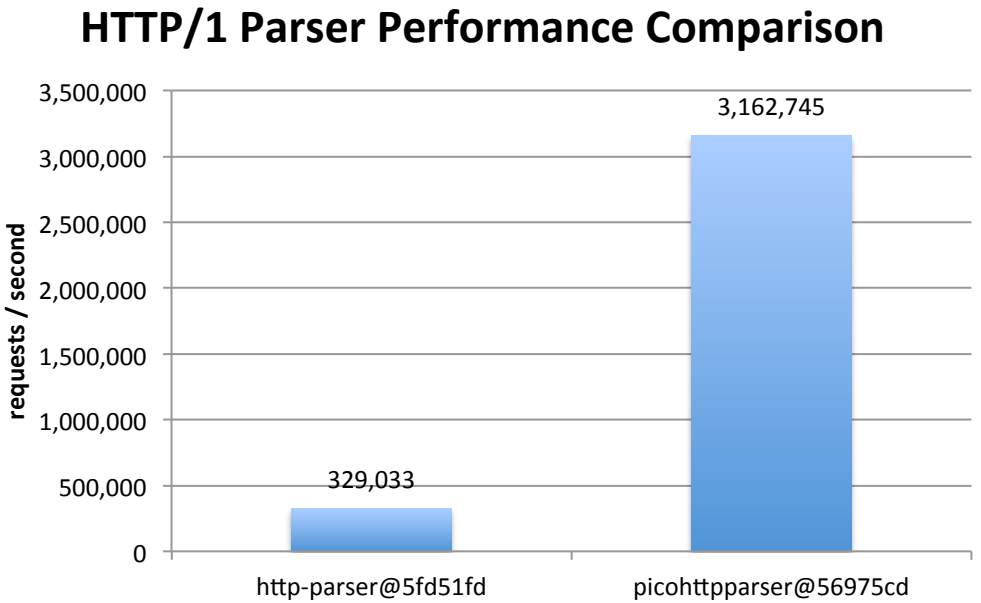
```
void *h2o_mempool_alloc(h2o_mempool_t *pool, size_t sz)
{
    (snip)
    void *ret = pool->chunks->bytes + pool->chunks->offset;
    pool->chunks->offset += sz;
    return ret;
}
```

- free
  - no code (as explained)

Parsing input

Parsing input

- HTTP/1 request parser may or may not be a bottleneck, depending on its performance
  - if the parser is capable of handling 1M reqs/sec, then it will spend 10% of time if the server handles 100K reqs/sec.



Parsing input (cont'd)

- it's good to know the logical upper-bound
  - or we might try to optimize something that can no more be faster
- Q. How fast could a text parser be?

Q. How fast could a text server be?

Answer: around 1GB/sec. is a good target

- since any parser needs to read every byte and execute a conditional branch depending on the value
  - # of instructions: 1 load + 1 inc + 1 test + 1 conditional branch
  - would likely take several CPU cycles (even if superscalar)
  - unless we use SIMD instructions

Parsing input

- What's wrong with this parser?

```
for (; s != end; ++s) {
    int ch = *s;
    switch (ctx.state) {
    case AAA:
        if (ch == ' ')
            ctx.state = BBB;
        break;
    case BBB:
        ...
}
```

Parsing input (cont'd)

- never write a character-level state machine if performance matters

```
for (; s != end; ++s) {
    int ch = *s;
    switch (ctx.state) { // ← executed for every char
    case AAA:
        if (ch == ' ')
            ctx.state = BBB;
        break;
    case BBB:
        ...
}
```

Parsing input fast

- each state should consume a sequence of bytes

```
while (s != end) {
    switch (ctx.state) {
    case AAA:
        do {
            if (*s++ == ' ') {
                ctx.state = BBB;
                break;
            }
        } while (s != end);
        break;
    case BBB:
        ...
```

Stateless parsing

- stateless in the sense that no *state* value exists
  - stateless parsers are generally faster than stateful parsers, since it does not have *state* - a variable used for a conditional branch
- HTTP/1 parsing can be stateless since the request-line and the headers arrive in a single packet (in most cases)
  - and even if they did not, it is easy to check if the end-of-headers has arrived (by looking for CR-LF-CR-LF) and then parse the input
    - this countermeasure is essential to handle the Slowloris attack

picohttpparser is stateless

- states are the execution contexts (instead of being a variable)

```
const char* parse_request(const char* buf, const char* buf_end, …)
{
  /* parse request line */
  ADVANCE_TOKEN(*method, *method_len);
  ++buf;
  ADVANCE_TOKEN(*path, *path_len);
  ++buf;
  if ((buf = parse_http_version(buf, buf_end, minor_version, ret)) == NULL)
    return NULL;
  EXPECT_CHAR('\015');
  EXPECT_CHAR('\012');
  return parse_headers(buf, buf_end, headers, num_headers, max_headers, …);
}
```

loop exists within a function (≒state)

- the code looks for the end of the header value

```
#define IS_PRINTABLE(c) ((unsigned char)(c) - 040u < 0137u)


static const char* get_token_to_eol(const char* buf, const char* buf_end, …
{
    while (likely(buf_end - buf >= 8)) {
#define DOIT() if (unlikely(! IS_PRINTABLE(*buf))) goto NonPrintable; ++buf
        DOIT(); DOIT(); DOIT(); DOIT();
        DOIT(); DOIT(); DOIT(); DOIT();
#undef DOIT
        continue;
    NonPrintable:
        if ((likely((uchar)*buf < '\040') && likely(*buf != '\011'))
            || unlikely(*buf == '\177'))
            goto FOUND_CTL;
    }
```

The hottest loop of picohttpparser (cont'd)

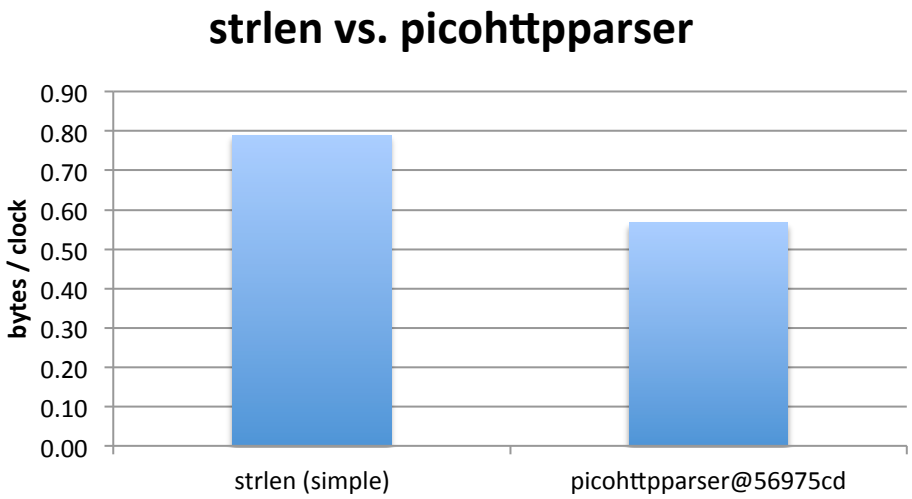
- after compilation, uses 4 instructions per char

```
movzbl  (%r9), %r11d
movl    %r11d, %eax
addl    $-32, %eax
cmpl    $94, %eax
ja      LBB5_5
movzbl  1(%r9), %r11d       // load char
leal    -32(%r11), %eax     // subtract
cmpl    $94, %eax           // and check if is printable
ja      LBB5_4              // if not, break
movzbl  2(%r9), %r11d       // load next char
leal    -32(%r11), %eax     // subtract
cmpl    $94, %eax           // and check if is printable
ja      LBB5_15             // if not, break
movzbl  3(%r9), %r11d       // load next char
…
```

strlen vs. picohttparser

- not as fast as strlen, but close

```
size_t strlen(const char *s) {
  const char *p = s;
  for (; *p != '\0'; ++p)
    ;
  return p - s;
}
```



- not much room left for further optimization (wo. using SIMD insns.)

picohttpparser is small and simple

```
$ wc picohttpparser.?
     376    1376   10900 picohttpparser.c
      62     333    2225 picohttpparser.h
     438    1709   13125 total
$
```

- good example of do-it-simple-for-speed approach
  - H2O (incl. the HTTP/2 parser) is designed using the approach

Stringification

Stringification

- HTTP/1 responses are in strings

```
sprintf(buf, "HTTP/1.%d %d %s\r\n", …)
```

- s(n)printf is known to be slow
  - but the interface is great
  - it's tiresome to write like:

```
p = strappend_s(p, "HTTP/1.");
p = strappend_n(p, minor_version);
*p++ = ' ';
P = strappend_n(p, status);
*p++ = ' ';
p = strappend_s(p, reason);
p = strappend_s(p, "\r\n");
```

Stringification (cont'd)

- stringification is important for HTTP/2 servers too
  - many elements still need to be stringified
    - headers (status, date, last-modified, etag, …)
    - access log (IP address, date, # of bytes, …)

Why is s(n)printf slow?

- it's a state machine
  - interprets the format string (e.g. "hello: %s") at runtime
- it uses the locale
  - not for all types of variables, but…
- it uses varargs
- it's complicated
  - sprintf may parse a number when used for stringifying a number

```
sprintf(buf, "%11d", status)
```

How should we optimize s(n)printf?

- by compiling the format string at compile-time
  - instead of interpreting it at runtime
  - possible since the supplied format string is almost always a string literal
- and that's qrintf

qrintf

- qrintf is a preprocessor that rewrites s(n)printf invocations to set of functions calls specialized to each format string
- qrintf-gcc is a wrapper of GCC that
  - first applies the GCC preprocessor
  - then applies the qrintf preprocessor
  - then calls the GCC compiler
- similar wrapper could be implemented for Clang
  - but it's a bit harder
  - help wanted!

Example

```
// original code (248 nanoseconds)
snprintf(buf, sizeof(buf), "%u.%u.%u.%u",
    (addr >> 24) & 0xff, (addr >> 16) & 0xff, (addr >> 8) & 0xff, addr & 0xff);

// after preprocessed by qrintf (21.5 nanoseconds)
_qrintf_chk_finalize(
  _qrintf_chk_u(_qrintf_chk_c(
      _qrintf_chk_u(_qrintf_chk_c(
          _qrintf_chk_u(_qrintf_chk_c(
              _qrintf_chk_u(
                  _qrintf_chk_init(buf, sizeof(buf)), (addr >> 24) & 0xff),
          '.'), (addr >> 16) & 0xff),
      '.'), (addr >> 8) & 0xff),
  '.'), addr & 0xff));
```

Performance impact on H2O

- 20% performance gain
  - gcc: 82,900 reqs/sec
  - qrintf-gcc: 99,200 reqs/sec.
- benchmark condition:
  - 6-byte file GET over HTTP/1.1
  - access logging to /dev/null

Timeout handling

Timeout handling by the event loops

- most event loops use balanced trees to handle timeouts
  - so that timeout events can be triggered fast
  - cons. is that it takes time to set the timeouts
- in case of HTTP, timeout should be set at least once per request
  - otherwise the server cannot close a stale connection

Timeout requirements of a HTTP server

- much more set than triggered
  - is set more than once per request
  - most requests succeed before timeout
- the timeout values are uniform
  - e.g. request timeout for every connection would be the same (or i/o timeout or whatever)

- balanced-tree does not seem like a good approach
  - any other choice?

Use pre-sorted link-list

- H2O maintains a linked-list for each timeout configuration
  - request timeout has its own linked-list, i/o timeout has its own, …
- how to set the timeout:
  - timeout *entry* is inserted at the end of the linked-list
    - thus the list is naturally sorted
- how the timeouts get triggered:
  - H2O iterates from the start of each linked-list, and triggers those that have timed-out

Comparison Chart

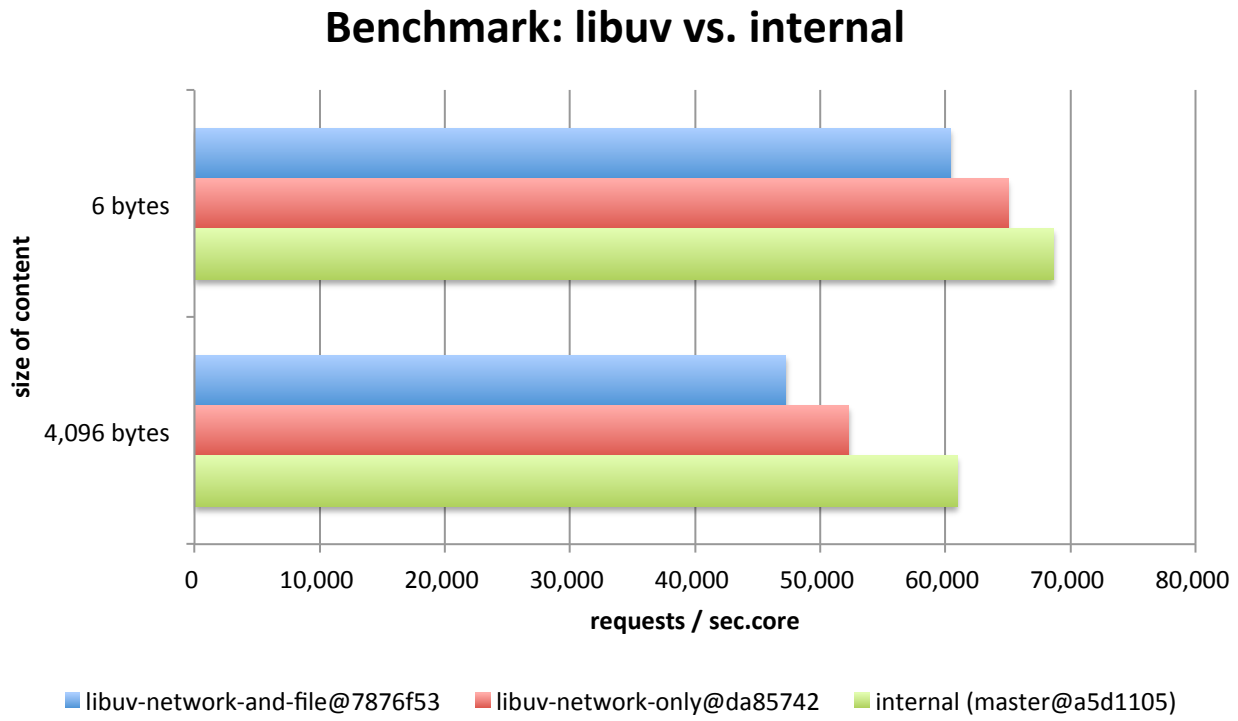
Operation (frequency in HTTPD)	Balanced-tree	List of linked-list
set (high)	O(log N)	O(1)
clear (high)	O(log N)	O(1)
trigger (low)	O(1)	O(M)

note: N: number of timeout entries, M: number of timeout configurations, trigger performance of list of linked-list *can* be reduced to O(1)

Miscellaneous

Miscellaneous

- **the entire stack of H2O is carefully designed (for simplicity and for performance)**
  - for example, the built-in event loop of H2O (which is the default for h2o), is faster than libuv



Writing H2O modules

Module types of H2O

- handler
  - generates the contents
    - e.g. file handler, proxy handler
- filter
  - modifies the content
    - e.g. chunked encoder, deflate
  - can be chained
- logger

Writing a "hello world" handler

```
static int on_req(h2o_handler_t *self, h2o_req_t *req) {
    static h2o_generator_t generator = {};
    static h2o_buf_t body = H2O_STRLIT("hello world\n");
    if (! h2o_memis(req->method.base, req->method.len, H2O_STRLIT("GET")))
        return -1;
    req->res.status = 200;
    req->res.reason = "OK";
    h2o_add_header(&req->pool, &req->res.headers, H2O_TOKEN_CONTENT_TYPE,
        H2O_STRLIT("text/plain"));
    h2o_start_response(req, &generator);
    h2o_send(req, &body, 1, 1);
    return 0;
}

h2o_handler_t *handler = h2o_create_handler( host_config, sizeof(*handler));
handler->on_req = on_req;
```

The handler API

```
/**
 * called by handlers to set the generator
 * @param req the request
 * @param generator the generator
 */
void h2o_start_response(h2o_req_t *req, h2o_generator_t *generator);
/**
 * called by the generators to send output
 * note: generator should close the resources opened by itself after sending the
final chunk (i.e. calling the function with is_final set to true)
 * @param req the request
 * @param bufs an array of buffers
 * @param bufcnt length of the buffers array
 * @param is_final if the output is final
 */
void h2o_send(h2o_req_t *req, h2o_buf_t *bufs, size_t bufcnt, int is_final);
```

The handler API (cont'd)

```
/**
 * an object that generates a response.
 * The object is typically constructed by handlers that call h2o_start_response.
 */
typedef struct st_h2o_generator_t {
    /**
     * called by the core to request new data to be pushed via h2o_send
     */
    void (*proceed)(struct st_h2o_generator_t *self, h2o_req_t *req);
    /**
     * called by the core when there is a need to terminate the response
     */
    void (*stop)(struct st_h2o_generator_t *self, h2o_req_t *req);
} h2o_generator_t;
```

Module examples

- Simple examples exist in the examples/ dir
- lib/chunked.c is a good example of the filter API

Current Status & the Future

Development Status

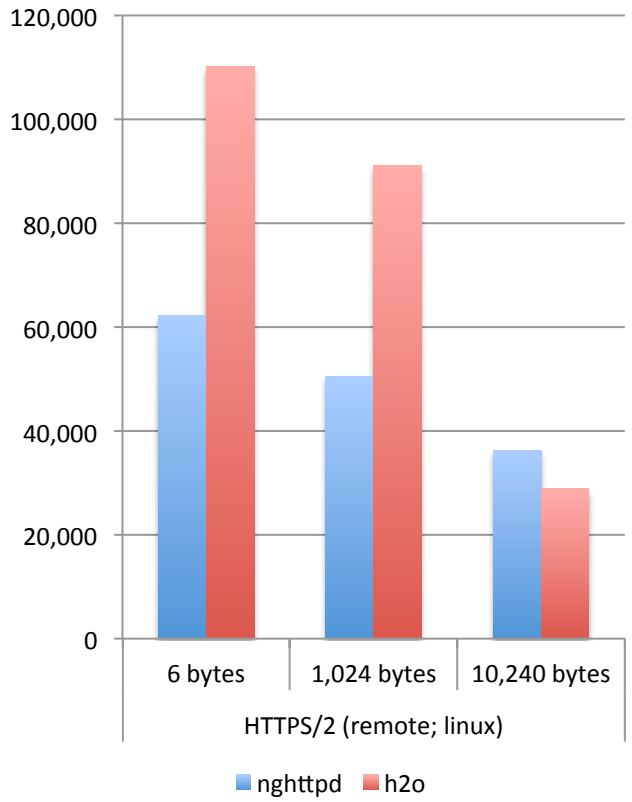
- core
  - mostly feature complete
- protocol
  - http/1 – mostly feature complete
  - http/2 – interoperable
- modules
  - file – complete
  - proxy – interoperable
    - name resolution is blocking
    - does not support keep-alive

HTTP/2 status of H2O

- interoperable, but some parts are missing
  - HPACK resize
  - priority handling
- priority handling is essential for HTTP/2
  - without, HTTP/2 is slower than HTTP/1 ☹
- need to tweak performance
  - SSL-related code is not yet optimized
    - first benchmark was taken last Saturday ☺

HTTP/2 over TLS benchmark

- need to fix the dropdown, likely caused by:
  - H2O uses writev to gather data into a single socket op., but OpenSSL does not provide scatter-gather I/O
    - in H2O, every file handler has its own buffer and pushes content to the protocol layer
      - nghttpd pulls instead, which is more memory-



Goal of the project

- to become the best HTTP/2 server
  - with excellent performance in serving static files / as a reverse proxy
    - note: picohttpserver and other libraries are also used in the reverse proxy implementation
- to become the favored HTTP server library
  - esp. for server products
  - to widen the acceptance of HTTP protocol even more

Help wanted

- looking for contributors in all areas
  - addition of modules might be the easiest, since it would not interfere with the development of the core / protocol layer
  - examples, docs, tests are also welcome
- it's easy to start
  - since the code-base is young and simple

Subsystem	wc –l (incl. unit-tests)
Core	2,334
Library	1,856
Socket & event loop	1,771
HTTP/1 (incl. picohttpparser)	886
HTTP/2	2,507
Modules	1,906
Server	573

Questions regarding HTTP/2

Sorry, I do not have much to talk

- since it is a well-designed protocol
- and in terms of performance, apparently binary protocols are easier to implement than a text protocol ☺
  - there's a efficient algorithm for the static Huffman decoder
    - @tatsuhiro-t implemented it, I copied
- OTOH I have some questions re HTTP/2

Q. would there be a max-open-files issue?

- according to the draft, recommended value of MAX_CONCURRENT_STREAMS is >= 100
- if max-connections is 1024, it would mean that the max fd would be above 10k
  - on linux, the default (NR_OPEN) is 1,048,576 and is adjustable
  - but on other OS?
- H2O by default limits the number of in-flight requests *internally* to 16
  - the value is configurable

Q. good way to determine the window size?

- initial window size (64k) might be too small to saturate the avaiable bandwidth depending on the latency
  - but for responsiveness we would not want the value to be too high
  - is there any recommendation on how we should tune the variable?

Q. should we continue to use CDN?

- HTTP/2 has priority control
  - CDN and primary website would use different TCP connection
    - means that priority control would not work bet. CDN and the primary website
- should we better serve all the asset files from the primary website?

Never hide the Server header

- name and version info. is essential for interoperability
  - many (if not all) webapps use the User-Agent value to evade bugs
  - used to be same at the HTTP/1 layer in the early days
- there will be interoperability problems bet. HTTP/2 impls.
  - the Server header is essential for implementing workarounds
- some believe that hiding the header improves security
  - we should speak that they are wrong; that security-by-obscurity does not work on the Net, and hiding the value harms interoperability and the adoption of HTTP/2

Summary

Summary

- H2O is an optimized HTTP server implementation
  - with neat design to support both HTTP/1 and HTTP/2
  - is still very young
    - lots of areas to work on!
    - incl. improving the HTTP/2 support
- help wanted! Let's write the HTTPD of the future!