Writing a Fast HTTP Parser

Kazuho Oku
DeNA Co., Ltd.
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.

![HTTP/1 Parser Performance Comparison (2014)](image-url)
How fast could a text parser be?

- 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?

```c
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.

```c
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

```c
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)

```c
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

```c
#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;
}
```

Writing a Fast HTTP Parser
The hottest loop of picohttpparser (cont'd)

- after compilation, uses 4 instructions per char

```assembly
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. picohttpparser

- not as fast as strlen, but close

```c
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.)
in 2015, we've did SIMD, got 2x speed

latest version uses SSE 4.2 insns; a fork that runs 2x faster using AVX2 insns. also exists (developed by CloudFlare)
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
conclusion

- text-based parser can be pretty fast
  - so that the overhead compared to binary-based approach is negligible
- we should not introduce binary-based structure for encoding headers in the future versions of HTTP, in the notion that it would be faster