Serving dynamic webpages in less than a millisecond

John Fremlin

2008 November 8

Contents

This talk is about a web-application framework I built.

A web-application framework in Lisp

- Designed for performance
- Still a work in progress
- Built from scratch
- Deals with everything from system calls to user sessions
- Integrated JavaScript generation (AJAX)
- Simple persistent in-memory database
- Made in Common Lisp (no C libraries)

Motivation

The combination of a fast dynamic webserver with modern webbrowser Javascript implementations is an untapped opportunity for ground-breaking interactive web-applications.

- Many fast static webservers exist: nginx, lighttpd
- But massively slower for dynamic content
- Using interpreted languages
- Horrible caching hacks (fragment caches, varnish, etc.)
- Small AJAX requests are increasingly useful

Demo: simple message-board

- Post messages to a message board
- Watch them appear immediately without reloading the page
- This is hard to scale (Twitter)

Implementation

- About 70 lines
- Uses AJAX
- Simple in memory database for messages
- Can be modified dynamically

Data structures

```
(defrecord message
  (forum-name :index t)
  text
  (author :index t)
  (time :initform (get-universal-time)))


(defmyclass (forum (:include simple-channel))
    name)

(defvar *fora* (list
                 (make-forum :name "Ubuntu")
                 (make-forum :name "Gentoo")
                 (make-forum :name "Debian")))
```

Website definition

```
(with-site (:page-body-start
            (lambda(title)
              (declare (ignore title))
              `(<div :class "header"
                     (<h1
                       (<A :href (page-link "/tlug")
                           :class "inherit"
                           (<span :style (css-attrib :color "red") "TLUG") " demo" ))
                     (output-object-to-ml (webapp-frame))))
            :page-head (lambda(title)
                         `(<head
                           (<title (output-raw-ml ,title))
                           (webapp-default-page-head-contents))))
  (defpage "/tlug" ()
    (webapp "Select forum"
      (webapp-select-one ""
                         *fora*
                         :display (lambda(forum) (<span (its name forum)))
                         :replace
                         (lambda(forum)
                           (webapp ()
                             (webapp-display forum)))))))
```

Rendering the data-structures

```
(my-defun forum 'object-to-ml ()
  (<div :class "forum"
        (<h3 (my name))
        (html-action-form "Post␣a␣message"
            (text)
          (make-message :forum-name (my name)
                        :text text
                        :author (frame-username (webapp-frame)))
          (my notify)
          (values))

        (<div :class "messages"
              (output-object-to-ml
               (datastore-retrieve-indexed 'message 'forum-name (my name))))
        (output-raw-ml (call-next-method))))


(my-defun message 'object-to-ml ()
  (<div :class "message"
        (<p (my text) (<span :class "message-attribution"
                             "␣by␣" (my author) "␣at␣" (time-string (my time))))))


(defun time-string (ut)
  (multiple-value-bind
        (second minute hour date month year day daylight-p zone)
      (decode-universal-time ut 0)
    (declare (ignore day daylight-p zone))
    (format nil "~4,'0D-~2,'0D-~2,'0D␣~2,'0D:~2,'0D:~2,'0D␣UTC"
            year month date hour minute second)))
```

Thoughts

Any questions?

Benchmarking the framework overhead

How many requests per second can be handled on one core?

- Request a page giving a name
- Reply with `<h1>Hello NAME</h1>` (properly escaping NAME)

Tests the overhead of the framework, excluding the database.

```
schedtool -a 1 -e ab -n 10000 -c100
http://localhost:3001/?name=TLUG
```

The advantage of my framework is that the complex work to determine the content to display can be done in a fast compiled language.

Ruby/C mongrel web-server

- Bare bones webserver that does not even parse the query string
- Often used for Ruby on Rails
- 1844.88 requests/sec

```
require 'mongrel'
require 'cgi'

class SimpleHandler < Mongrel::HttpHandler
  def process(request, response)
    response.start(200) do |head,out|
      head["Content-Type"] = "text/html"
      name = CGI::escapeHTML(CGI::parse(
                         request.params['QUERY_STRING']
                         )['name'].first)
      out.write("<h1>Hello #{name}</h1>")
    end
  end
end

h = Mongrel::HttpServer.new("0.0.0.0", "3000")
h.register("/", SimpleHandler.new)
h.run.join
```

PHP

- Lighttpd with FastCGI
- No code cache
- Logging disabled
- 3174.52 requests/sec

```
<h1>Hello <?= $_REQUEST['name'] ?></h1>
```

My implementation

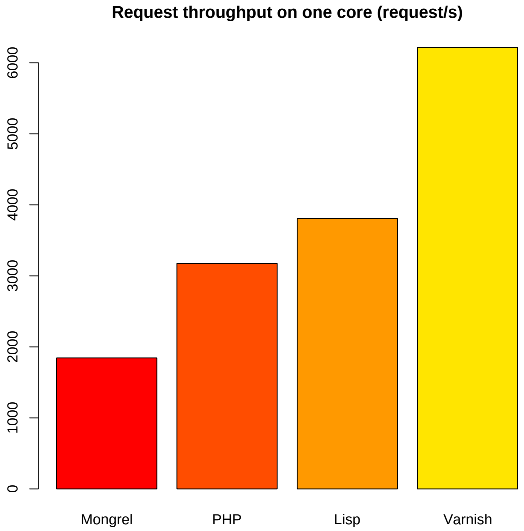
- Automatic escaping
- Plenty of parentheses
- 3806.90 requests/second

```
(defpage "/test" (name)
  (<h1 "Hello␣" name))

(launch-io 'accept-forever
           (make-con-listen :port 3000)
           'http-serve)

(event-loop)
```

Varnish

- Cache
- Does not do a dynamic request
- Heavily optimized
- A guiding figure for the maximum possible speed, assuming zero cost for the dynamic request
- 6217.97 requests/second

Results


Request throughput on one core (request/s)
0
1000
2000
3000
4000
5000
6000
Mongrel
PHP
Lisp
Varnish

Thoughts

The framework overhead is not too important, provided it is reasonable. *The advantage of my framework is that the complex work to determine the content to display can be done in a fast compiled language.*

History of high performance HTTP under Linux

In 1999, Microsoft had the fastest web-server with IIS.

- Mindcraft benchmark (1999 April)
- TUX, a kernelspace webserver by Ingo Molnar at RedHat (2000 July)
- Record breaking SPECWeb99 scores. Twice as fast as IIS

By 2001 May, IIS was again slightly faster than TUX
Performance similar to TUX 2 can now be achieved outside the kernel

- Lighter context-switches
- Less copying for IO
- TCP cork

Zero-copy IO

- Copying data wastes time
- Simple caches are made useless
- sendfile(2) solves this for files from disk
- writev(2) helps for dynamic content
- TCP checksum

Lighter context switches

- Linux always had fast syscalls but the pthreads implementation was very slow
- Native POSIX Thread Library (futexes)

But user-level threading will generally be faster

- Enabled with poll(2)
- Traditionally used by IRC daemons
- But does not scale to large numbers of connexions

poll does not scale

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
  int    fd;          /* file descriptor */
  short events;       /* requested events */
  short revents;      /* returned events */
};
```

Poll makes an $O(n)$ data transfer to the kernel for every wait, where n is the number of connexions

```
int epoll_wait(int epfd, struct epoll_event *events,
  int maxevents, int timeout);
struct epoll_event {
  uint32_t     events;      /* Epoll events */
  epoll_data_t data;        /* User data variable */
};
int epoll_ctl(int epfd, int op, int fd, struct epoll_eve
int epoll_create(int size);
```

epoll does scale

- Similar to
  - /dev/epoll on Solaris
  - kqueue on FreeBSD
- Allows edge-triggering
- Annoyingly cannot be used with disk files
- And AIO cannot be used for network sockets

Handling many simultaneous connections

- One process per connection: slow
- One OS thread per connection: better
- Multiplexing connections inside one thread: fast
    - select: old
    - poll: better
    - epoll: fastest

TCP cork

- Avoid sending out partial packets for the HTTP header
- Even if it takes some time to generate the body
- Very important for TUX2

But actually heavily detrimental to performance for me

Engineering decisions

- Implemented entirely in Common Lisp
- No C libraries
- Entirely Linux specific
- Can run at reasonable speeds on SBCL and ClozureCL

Network module

- Many connexions per thread
- Cannot block in any protocol handler
- Uses code-transformer to generate state-machines from code written in an imperative style
- Key system calls: epoll, read, writev

HTML generation module

Generates chains of strings to send to writev(2) and offers bonus compile-time typo checking

- Misspelled attributes
- Misplaced tags (for example `<li>` in a `<p>`)
- Misspelled CSS properties

Parenscript

A library for writing JavaScript in Lisp. I did not develop this.

- Advanced code generation with Lisp-style macros
- Generates predictable, readable JavaScript
- Easy to debug with in Firebug
- Modified to do more work at compile time
- Very handy, because code can be shared between the server-side and client-side (browser)

cl-irregsexp

- Regular expression engine
- Unusual syntax
- Fast for some things
- Generates native code

Conclusion

This project was a huge waste of time!
And it’s not finished.

Final demo

The combination of a fast dynamic webserver with modern webbrowser Javascript implementations is an untapped opportunity for ground-breaking interactive web-applications.
Any ideas?