Developing high-performance network servers in Lisp

Vladimir Sedach <vsedach@gmail.com>

Overview

- Servers and protocols
- Strings, character encoding, and memory
- Concurrency and asynchrony
- Overview of Common Lisp HTTP servers employing these techniques

(Almost) no new ideas

- Basic considerations described by Jeff Darcy:
- http://pl.atyp.us/content/tech/servers.html

What's a server, anyway?

- Get input
- Process input
- Write output

How is that different from any other program?

- Many clients
- High-latency connections
- Need to serve each client quickly
- Need to serve many clients simultaneously

Protocols

- ”Process” part of Input-Process-Output is application specific
- Server is focused on I/O
- I/O is protocol-specific

HTTP

- 1.0: One request-response per TCP connection
- 1.1: Persistent connection, possibly multiple (pipelined) request-responses per connection
- Comet: Server waits to respond until some event occurs (faking two-way communication)

Cache is King

- Memory access is expensive
- Memory copying is expensive

Character encoding

- How does your Lisp do it?
- SBCL: 32 bits per character
- read() returns vector of bytes

UTF-8

- Use UTF-8 everywhere
- Do everything with byte vectors (DIY character encoding)
- No need to copy/re-encode result of read()
- Do encoding of output at compile-time with (compiler) macros

Vectored I/O

- Access and copying is slow, so concatenation is slow
- System calls also slow
- Use writev() to output a list of byte vectors in one system call

Input

- Vectored input a pain, not worth it for HTTP
- Reuse input buffers!
- In some Common Lisp implementations (SBCL), think about declaring inline and ftype for FFI read() etc input functions to cut down on consing (small overhead but adds up)

Demultiplexing

- Threads
- Asynchronous IO

Threads

- Thread Per Connection (TPC)
- Thread blocks and sleeps when waiting for input
- Threads have a lot more memory overhead than continuations
- Passing information between threads expensive – synchronization and context switch
- But can (almost) pretend we're servicing one client at a time

Asynchronous I/O

- I/O syscalls don't block
- When cannot read from one client connection, try reading from another
- Basically coroutines
- epoll/kqueue interfaces in Linux/BSD provide efficient mechanism that does readiness notification for many fds

(Not So) Asynchronous I/O

- Most AIO HTTP servers not really AIO
- Just loop polling/reading fds until a complete HTTP request has been received, then hand it off to a monolithic process_request() function
- What happens when process_request() blocks on DB/file access?
- What happens with more complicated protocols where input and processing are intertwined?

Handling AIO Events

- Have to resort to state machines if you don't have call/cc
- For true AIO, need to involve state handling code with application code (even for HTTP for things like DB access)
- Not feasible with state machines
- Trivial with first-class continuations

Thread pools

- Typical design – one thread demultiplexes (accept()), puts work in a task queue for worker threads
- Context switch before any work can get done
- Synchronization costs on task queue

Leader-Follower (LF)

- http://www.kircher-schwanninger.de/michael/publications/lf.pdf
- (let (fd)
- (loop (with-lock-held (server-lock)
- (setf fd (accept server-socket)))
- (process-connection fd)))
-
- On Linux and Windows, accept() is thread-safe – no need for taking server-lock

Which one is better?

- Paul Tyma (Mailinator, Google) says thread-per-connection handles more requests
- http://paultyma.blogspot.com/2008/03/writing-java-multithreaded-servers.html
- For many simultaneous connections, AIO consumes much less memory while making progress on input
- In SBCL, AIO seems to win

Antiweb

- http://hoytech.com/antiweb/
- Async (predates both nginx and lighttpd)
- Connections managed by state machine
- Multi-process, single-threaded
- Doesn't use Lisp strings (UTF-8 in byte vectors)
- Vectored I/O (incl support for HTTP pipelining – send multiple responses in one writev())

TPD2

- http://common-lisp.net/project/teepeedee2/
- Async
- Connections managed by continuations (code transformed with cl-cont)
- Single-process, single-threaded
- Doesn't use Lisp strings (UTF-8 in byte vectors)
- Vectored I/O

TPD2 goodies

- Comes with mechanism for vectored I/O (sendbuf)
- Comes with macros for UTF-8 encoding at compile-time (incl. HTML generation macros)
- Really fast on John Fremlin's benchmark – 11k requests/sec vs 8k r/s for nginx and embedded Perl (http://john.freml.in/teepeedee2-c10k)

Rethinking web servers

- Look at HTTP as a protocol
- Want to respond to initial request quickly
- For persistent connections, there may or may not be another request coming, but the connections themselves pile up because of long keepalive times
- Comet and long polling – assume huge number of connections sitting idle waiting for something to happen (ex - Web2.0 chat)

HTTP DOHC

- http://github.com/vsedach/HTTP-DOHC
- (Needs repository version of IOLib with my patches, available on IOLib mailing list)
- Hybrid TPC/AIO architecture (new)
- Single process, multi-threaded
- Doesn't use Lisp strings (UTF-8 in byte vectors)
- Not finished yet
- Not as fast as TPD2, faster than nginx

HTTP DOHC architecture

- LF thread pool demultiplexing accept() and immediately handling the initial HTTP request
- If HTTP connection is persistent, hand it off to second LF thread pool demultiplexing epoll/kqueue to handle (possible) second and subsequent requests
- For now the second thread pool still does TPC. Can make it do AIO WOLOG

Why separate LF pools?

- Combining multiple event sources into one results in an inverse leader/task-queue/worker pattern
- Planning to look at SEDA for ideas on moving threads around between pools to handle load

HTTP DOHC TODO

- Plan to make interface as compatible as reasonable with Hunchentoot 1.0
- Planned third LF thread pool to demultiplex synthetic server-side events to handle Comet requests
- File handling
- SSL
- Gzip encoding

Conclusions

- Good server design depends on protocol