How **not** to write network applications

(and how not to use them correctly..)

Adrian Chadd <adrian@FreeBSD.org>
Overview

• A simple overview - including HTTP basics
• A few “bad” examples, notably from Squid/Apache - and what they’ve subsequently done
• An “ok” example - notably lighttpd
• “good” examples - memcached, varnish
• What is libevent?
Overview (ctd)

- Latency, bandwidth delay product, and scheduling network IO
- Why does disk IO matter?
- Summary
Introduction

• Writing network applications is easy
• Writing efficient network applications is less easy
• Writing efficient, scalable network applications is even less easy
• Predicting your real-life workloads and handling that is difficult
Lessons learnt, #1

• High-performance network applications needs clue
  • Coding clue
    • Algorithm choices, structure
  • Hardware clue
    • How fast can you push what
    • Gathering/Interpreting profiling
Lessons learnt, #1

- Operating system clue
  - Best way to schedule stuff
  - Worst ways to schedule stuff
  - Profiling!

- Networking clue
  - “speed of light”
  - TCP/UDP behaviour
Lessons learnt, #1

- Protocol clue
  - How does the protocol work
  - Client <-> Server communication
  - Client behaviour, Server behaviour
  - How this ties into the network
An example: HTTP

- HTTP is strange

- A large variance in usage patterns, client/servers, traffic patterns, software versions, network behaviour..

- Small objects
  - < 64k
  - will never see TCP window size hit maximum during initial connection lifetime
An example: HTTP

- Large objects
  - Well, >64k really
  - Will start to hit congestion and back-off limits
- Throughput variations are perceived by end-user
  - versus small objects - request/reply rate dictates perceived speed
An example: HTTP

- But there’s more!
  - HTTP keepalives affect TCP congestion
  - HTTP pipelining influences perceived request speed on small objects
  - Clients and servers have differently tuned TCP stacks...
    - ..“download accelerators”, anyone?
Apache: History!

- The pre-fork web server
- Internals *should’ve* been clean because of this
- Handled high-req rate poorly
- Handled high numbers of concurrent connections poorly
- Flexible enough to run a variety of processing modules - php, python, perl, java..
Apache: History!

- Why did it perform so poorly under load?
  - Memory use - each connection == 1 process; SSL/PHP/Python/etc overheads
  - .. even if the request didn’t require any of that
  - scheduling 30,000 concurrent processes == hard (Jeff: is it that bad nowadays?)
  - small amount of paging == death
Apache 2: Revolution

- Decided to abstract out the dispatching runtime - thread pool, pre-fork
  - To handle varying platform support, incl. Windows, Netware
- Abstracted out the socket polling where relevant - select, poll, kqueue, epoll, etc
- User can select which dispatcher (MPM) they wish to use at compile/install time
Apache 2: MPM

• Quite a few MPM modules for scheduling work
  • Traditional prefork
  • Process + thread worker module
  • Thread-only worker modules (Netware)
  • Something windows-specific
Apache 2: Performance

- Pre-fork: same as apache 1
- Worker thread models:
  - network IO only? It should be fast enough for you
  - Disk IO too? Things get scary: the worker thread pool begins to grow!
- thread seems to scale (as a proxy) to >10000 concurrent connections
Apache 2: Modern Use

- Split up different services - static, dynamic, application

- Configure a front apache (running thread MPM) as a proxy; “route” content to applicable backend

- Static content? Don’t waste memory on PHP.

- PHP/etc content? Don’t stall static content serving
Squid: History

- Squid: its been around a while
- It's not as bad as people make it out to be
- It's getting better as I find free time
- Compared to modern proxies, it's slower..
  - .. but it handles a wide cross-section of traffic loads (except “lots of traffic” ..)
  - .. lots of traffic defined at ~ 1000 req/sec and about 200mbit of mixed traffic
Squid: internals

• Single process/thread event loop for everything but disk IO
• Non-blocking network IO
• Has grown kqueue/epoll/etc support
• Uses threads/processes to parallelise blocking disk IO
• Attempts to mitigate overload conditions where humanly possible (ie: where I find them)
Squid: what's wrong?

- Far too much code..
  - ~ 25 functions account for 40% of CPU
  - ~ 500 functions account for the other 60% of CPU (userland)
- IO done in small amounts
  - Disk IO - 4k
  - Network IO - 16k
- This isn’t as bad as you think.. read on
**Squid: what's wrong?**

CPU: Core 2, speed 2194.48 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (Unhalted core cycles) count 100000

samples % image name symbol name
216049 6.5469 libc-2.7.so memcpy
115581 3.5024 libc-2.7.so _int_malloc
103345 3.1316 libc-2.7.so vfprintf
85197 2.5817 squid memPoolAlloc
64652 1.9591 libc-2.7.so memchr
60720 1.8400 libc-2.7.so strlen
Squid: what's wrong?

- Codebase has grown organically
- Squid-cluey programmers were hired by Akamai, etc - suddenly no-one was working on performance
- Ten + years of features added on top of poor structural base, and HTTP/1.1 still hasn’t appeared..
- .. but the poor structure is now looking better
### Squid: network IO?

#### (ACCELERATOR)

**HTTP I/O**

- Number of reads: 19463301

**Read Histogram:**

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1- 1</td>
<td>5194</td>
<td>0%</td>
</tr>
<tr>
<td>2- 2</td>
<td>4675</td>
<td>0%</td>
</tr>
<tr>
<td>3- 4</td>
<td>1588</td>
<td>0%</td>
</tr>
<tr>
<td>5- 8</td>
<td>10412</td>
<td>0%</td>
</tr>
<tr>
<td>9- 16</td>
<td>351771</td>
<td>2%</td>
</tr>
<tr>
<td>17- 32</td>
<td>89452</td>
<td>0%</td>
</tr>
<tr>
<td>33- 64</td>
<td>63398</td>
<td>0%</td>
</tr>
<tr>
<td>65- 128</td>
<td>81808</td>
<td>0%</td>
</tr>
<tr>
<td>129- 256</td>
<td>337836</td>
<td>2%</td>
</tr>
<tr>
<td>257- 512</td>
<td>412245</td>
<td>2%</td>
</tr>
<tr>
<td>513- 1024</td>
<td>928914</td>
<td>5%</td>
</tr>
<tr>
<td>1025- 2048</td>
<td>14296942</td>
<td>73%</td>
</tr>
<tr>
<td>2049- 4096</td>
<td>1731657</td>
<td>9%</td>
</tr>
<tr>
<td>4097- 8192</td>
<td>808069</td>
<td>4%</td>
</tr>
<tr>
<td>8193-16384</td>
<td>205358</td>
<td>1%</td>
</tr>
<tr>
<td>16385-32768</td>
<td>60013</td>
<td>0%</td>
</tr>
</tbody>
</table>

#### (PROXY)

**HTTP I/O**

- Number of reads: 3087754

**Read Histogram:**

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1- 1</td>
<td>11327</td>
<td>0%</td>
</tr>
<tr>
<td>2- 2</td>
<td>208</td>
<td>0%</td>
</tr>
<tr>
<td>3- 4</td>
<td>1211</td>
<td>0%</td>
</tr>
<tr>
<td>5- 8</td>
<td>617</td>
<td>0%</td>
</tr>
<tr>
<td>9- 16</td>
<td>1421</td>
<td>0%</td>
</tr>
<tr>
<td>17- 32</td>
<td>3400</td>
<td>0%</td>
</tr>
<tr>
<td>33- 64</td>
<td>6079</td>
<td>0%</td>
</tr>
<tr>
<td>65- 128</td>
<td>14680</td>
<td>1%</td>
</tr>
<tr>
<td>129- 256</td>
<td>20808</td>
<td>1%</td>
</tr>
<tr>
<td>257- 512</td>
<td>57378</td>
<td>2%</td>
</tr>
<tr>
<td>513- 1024</td>
<td>2931775</td>
<td>95%</td>
</tr>
<tr>
<td>1025- 2048</td>
<td>25183</td>
<td>1%</td>
</tr>
<tr>
<td>2049- 4096</td>
<td>3767</td>
<td>0%</td>
</tr>
<tr>
<td>4097- 8192</td>
<td>4061</td>
<td>0%</td>
</tr>
<tr>
<td>8193-16384</td>
<td>5839</td>
<td>0%</td>
</tr>
<tr>
<td>16385-32768</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>
Squid: Network IO?

• Talking over a LAN != Talking over a WAN
• Larger socket buffers == faster throughput
  • But only up until bandwidth delay!
• Larger socket buffers also == wasted RAM
• Choose socket buffer size based on required throughput and concurrency, based on client delay.
  • .. which can vary, so its tricky ..
Theoretical: $\leq 4k$ bufs
Theoretical: $\leq 32k$ bufs
Socket buffers (again)

- So socket buffer sizes are a tradeoff
  - eg: 10,000 4k socket buffers: 40 megabytes
  - eg 10,000 32k socket buffers: 320 megabytes
  - Double that (at least) if the application buffers in-flight data until the kernel says its sent!
Squid: Disk IO

• Don’t use one-file-per-object for small, frequently accessed files
  • If you do, at least pretend to dump related objects in the same directory
  • open/close metadata overheads are high
  • If you’re unlucky, >2 seeks to open a file that isn’t in VM/buffer cache
  • .. and then the IO is done in 4k chunks
Squid: 4k disk IO?

- **Transfer rate at 32k (18gig 10krpm SCSI)**
  Runtime: 41.32 seconds, Op rate: 247.84 ops/sec, Avg transfer rate: 8121367.38 bytes/sec

- **Transfer rate at 4k (18gig 10krpm SCSI)**
  Runtime: 32.27 seconds, Op rate: 317.28 ops/sec, Avg transfer rate: 1299566.81 bytes/sec

- ops/sec drop by 22%; transfer rate up by 6x

- need to squeeze small objects into larger blocks on disk and increase IO size
Squid: logging

- It *did* use stdio calls for logging
- .. which may block on buffer flush
  - anecdotally, topping out the logging performance at ~ 300 req/sec
- Current logging code: log to memory buffer; send buffer over pipe() to helper process
- Later plans will turn this into a thread
- Limited by Squid: can log ~ 4000 req/sec to disk with no service impact
Squid: reply sizes

• Like the object histogram, actual reply sizes (and the time length to serve them) varies greatly
  • Forward proxy: mix of small and large
  • Accelerator: may be a mix; may be just small, may be just large, may be both
  • If you’re clever, you can handle all of these cases efficiently enough
    • .. or you can assume everyone is local..
Squid: reply sizes

- Sample 1: Forward proxy
  - < 8k - 314871
  - < 64k - 2448235
  - < 256k - 333
  - < 1M - 6761
  - < 32M - 20874
  - > 32M - 3132

- Most requests are < 64k; with a secondary small peak between 1M / 32M
Sample 2: (last.fm; used with permission)

- < 8k - 3249802
- < 64k - 5618618
- < 256k - 1357
- < 1M - 33407
- < 32M - 88592
- > 32M - 11511

Again, most are <64k; ~100k (~1.2%) are >32M

What are the implications of these?
Squid: reply sizes

• Those large replies will be streaming replies, either from disk or from another server

• Much more data transmitted!

• Long-held connections, potentially filled TX socket buffer

• Transmitting these should not interfere with small object replies

• ..and for the most part, Squid handles that dichotomy fine
Squid: load shedding

- At some point you’ll receive more requests then you can handle
- You need to gracefully(ish) handle this so the service doesn’t spiral into death
- Squid does this in a number of places
  - Too many connections? Stop accept()’ing
  - Too much disk IO? Stop disk HITs; fail to MISSes
Squid: accept() rates

• Balance accepting new connections and handling existing connections

• More difficult with poll()! (i.e., how often to run poll() over the incoming vs all sockets)

• In the past - full accept queue -> ignore new requests

• Currently (AFAICT) - full accept queue -> RST new requests

• Impacts service; impacts SLB logic
Squid: Disk IO

• Overloaded disk IO queue?
  • First: Turn object creates into create fails; limit hits to memory only
  • Then: Turn object reads into fails; limit hits to memory only - generally turn into temporary MISS -> backend fetch
  • Problem: increased backend load
    • .. and this can also cause your service to spiral down into death
Lighttpd: New Stuff

• The Ruby crowd loves this thing for some reason

• Isn’t a HTTP server so much as a “HTTP content router”

• Save a few things (eg static, flv); all complicated stuff is done via fastcgi back-ends

• Attempted to handle sendfile() where appropriate
lighttpd: internals

• Again - select/kqueue/poll/epoll style event loop with callbacks

• Monolithic process - SMP implemented as simply running >1 process
  • Which works very well for what lighttpd does

• Attempts to schedule “IO operations” internally which map to a variety of options
  • read, readv or sendfile, for example
lighttpd: what’s right

• The majority of complicated behaviour is implemented through fast-cgi modules
  • I.e., lighttpd doesn’t run PHP, etc in its own process

• This frees up lighttpd to be a HTTP content router to “other” things locally and/or over the network

• It just happens that it also serves static content quite well
lighttpd: what's wrong

• “Chunk” interface - A list of “chunks” to write to the client

• A “chunk” could be memory, disk, another network socket

• “disk” chunks would be read/sendfile ()'ed as needed..
  • .. and the whole process stopped if the read needed to block.

• Apparently fixed in later versions!
lighttpd: anecdotally

- Feedback from various teams inside a large content provider
- Lighttpd doing straight static replies:
  - ~ 1k: ~10,000 req/sec per CPU
  - ~ 2k: ~8000 req/sec per CPU
  - ~ 4k: ~5000 req/sec per CPU
  - > 8k: about the same speed as Squid
    - 4k, 5000 req/sec => 200mbit / sec
Varnish

- (Hi PHK!)
- Initially I had a lot to talk about, but my data has fallen through from third parties
Varnish

• A good example of how far you can push hardware and software

• A bit workload-specific: handles small objects well; much larger objects not so well

• Anecdotal evidence about handling lots of slow clients poorly (this is what I wanted data about!)
Varnish: internals

- (Insert PHK’s slides from last year here)
- Pool of worker threads
- Network/VM IO done sync, not async
- Parallelism through worker threads
- Good pthread locking, efficient parsing, efficient data exchange, doesn’t abuse memory allocator, VCL is shiny
Varnish: internals

• Instead of complicated hard-coded rules (à la Squid and most other things), forwarding and caching logic is implemented in VCL

• Which is translated into C and inserted into varnish at runtime

• Reliant on scatter-gather IO (good!) and VM system (not so good, see below)
Varnish: in production

- Works great for some
  - Hot workload fits in RAM; small objects? Fantastic
- Anecdotally, doesn’t work great for others
  - Slow backend w/ popular objects? Not so good. (Squid -> “collapsed forwarding”)
  - Slow clients/servers -> not so good
Varnish: VM?

- Varnish uses the VM system quite extensively
- The VM system is great at the average, but needs to be “taught” about HTTP access patterns to optimise disk throughput
- Eg: pack small objects into contiguous pages
- Eg: do IO in larger parts to save on disk ops
Varnish: the “good”

• Scales well across multiple CPUs
• Handles its workload very well
  • (i.e., puts other proxies to shame)
• Does stuff “differently” (in a good way)
  • Eg - logging, statistics reporting
  • VCL - don’t hard-code your application logic!
Memcached

• Or, as I like to call it, “mysqlcached”
• A memory object cache for storing and retrieving “stuff”
• “stuff” is generally SQL queries, but can be whatever the heck you want
Memcached: Internals

- Started as a Squid-like single process async event loop
- First time I saw it: it used libevent
- A couple years ago? - threaded
  - N threads, one per CPU
  - One thread handles incoming connections
  - All threads: handle actual work
Memcached: scaling

- It scales quite well..
- .. but it isn’t a complicated program!
- Memcached scaling is generally limited by OS parallelism - FDs, socket, TCP, UDP, IP
- Doesn’t need to schedule disk IO; all operations are memory based
Memcache: issues

- Similar to Squid/Varnish: small objects pack badly
- Apparently(!) Memcache tries to pack objects using 32 bit pointers in 64 bit environment
- Squid - 160 byte StoreEntry, 70 odd byte MD5; 30 odd byte MemObject; 4k object granularity
  - Memory wastage on small objects
Libevent?

- Libevent is a simple(!) library for scheduling network IO events across UNIX platforms
- Implements poll, select, kqueue, epoll, /dev/poll, solaris event ports
- (and Windows; but thats a different story)
- Basic threading support - run multiple event queues, one per thread
How is libevent used?

- Create queue - `event_base * event_init();`
- Run the queue - `event_base_loop (event_base *)`;
- Setup events - `event_set(event *, fd, what, callback, data)`
- Throw event into a queue - `event_base_set()`
- Schedule event - `event_add(event *, timeval *)`
Does libevent scale?

- Scales well across multiple CPUs - each libevent queue runs separately
- Event registration isn’t $O(1)$ - uses trees for registering timer/immediate events in priority/order
  - A “derivative” libevent tries to avoid this overhead
Trouble with Libevent

- Standard UNIX problem - inter-thread communication
- Thread sleeps on poll/select/kqueue/etc; how does another thread wake it?
- “portable” method - create pipe; write byte to “wake” up destination thread to check message queue
- Each UNIX has a different way of solving this!
How low can you go?

- A simple libevent-based TCP proxy
  - `accept()` connection, `connect()` to another; shuffle data
- CPU parallelism by using one thread per CPU
- Core 2 Duo desktop: E2200
- Variable socket sizes; variable concurrency
- How far can things be pushed?
TCP Proxy: One thread

- One userland thread
- One kernel thread for network device IO
  - Can split that into device/netisr threads
- Throughput: ~4kbyte objects; ~400mbit/sec; 12000 req/sec - 24,000 sockets/sec
- One CPU maxed userland; other CPU mostly maxed doing device/netisr
TCP Proxy: two threads

- Two userland threads
- Same setup
- Only incremental improvement - 500mbit; slightly more requests/sec
- Both CPUs at 100%
- Why?
TCP Proxy: contention

- One particular area of contention:
  - TCP PCB processing
  - Robert/Kris will be working on this
- Userland CPU breakdown:
  - <5% userland CPU both CPUs; so the userland is fine
  - Is it “doing” things efficiently?
TCP Proxy: buffer size

- What happens if you up the socket buf size?
- (And what happens if you up the transaction size?)
  - Transaction size: higher throughput; approaching 800mbit FDX
  - Socket buffer size: no appreciable difference on LAN
  - Need to model WAN traffic a little better!
Bandwidth Delay

- this isn’t just a problem on the WAN
- LAN’s have similar issues with gige/10ge pipesize
- In summary - you end up having to pipeline
- .. why would you need to pipeline on a LAN?
- (eg - NFS)
NFS and Delay

• Say, 4k transactions over the wire
• How can you get gigabit speed with 4k transactions?
  • 100 megabytes/sec / 4kbyte/sec => ~25k packets a second
  • Each transaction: 0.00004 sec (0.04 msec)
• If your transaction for 4k block > 0.04msec, you won’t saturate gigabit ethernet
NFS, Delay, real-world

• Computation cluster serving data over NFS
• Legacy fortran code, \( \sim 1 \text{ kbyte data chunking} \)
• Bad throughput!
  • CPU wasn’t maxed
  • Disks weren’t maxed
  • Network wasn’t maxed
  • ...?
NFS, Delay, Real world

- Problem is due to NFS transaction latency!
Disk IO ..?

• Lots of applications do disk IO to push out to the network

• Think about latency on disk IO + latency on network IO -> effective transfer rates

• UNIX network IO - traditionally sync
  • POSIX AIO makes this less painful
  • Faked using Threads/Processes
Scheduling Disk IO

- How it's done in Squid:
  - `aio_read(fd, buf, size, callback, cbdata)`
  - ... buffer is returned
  - `.event_add(socket write event, timeout)`
  - .. socket is ready
  - `write(sockfd, buf, size)`
Scheduling disk IO

• The problems!
  • Standard UNIX read/write involves a kernel copyin/copyout, which takes quite a bit of time
  • POSIX AIO in FreeBSD should make this much less painful - shouldn’t copy disk data
  • Prefetching or no-prefetching?
Disk IO: prefetching?

- How much data can you pre-fetch?
  - Balance between reading slightly more data from disk, and how much RAM in your box (and buffer cache)

- `mmap()`?
  - Again, potentially blocking!
  - You have to manually lock pages or they may even be removed underneath you.
Disk IO: sendfile

- Sendfile is a “pretty word”
- In essence - glue together a disk fd and a socket fd; ask kernel to do the heavy lifting for you without copying
- You avoid two trips user->kernel for the disk read, then the socket write
- Traditionally: blocking only; so you need threads to run the sendfile context()
- (ie, one reason varnish is what it is..)
Summary

• Writing efficient, scalable network applications is hard
• Understand what you’re trying to do
• Understand how you can do it
• Understand your protocol, hardware, software
• And above all - assume users will do dirty things with it that you don’t expect!
Questions?
Thankyou!

Adrian Chadd <adrian@FreeBSD.org>