Squid: Storage Stuff

Adrian Chadd <adrian@squid-cache.org>

Disclaimer!

- A lot of the "science" is sitting on currently-off servers locked away in a datacentre I couldn't access before the talk
- .. as such, please take this with a grain of salt until the science can be backed up.

Storage: overview

- Two main “types”: UFS, COSS
- UFS breakdown into three IO types
  - ufs
  - aufs
  - diskd
- COSS breakdown into POSIX AIO and AUFS

Storage: objects

- Entire object is stored on disk as received from server-side..
- .. with a little bit of TLV-encoded metadata prepended
- No separation between reply headers and reply body - its just “reply data” as far as the store is concerned

Storage: UFS

- One file per object
- Semantics: open, create, close, unlink, read, write
- Massive code duplication between UFS, AUFS and DISKD (thanks to my early Squid development efforts :/); mostly unwound in Squid-3

Storage: COSS

- Uses large file / raw device to implement small object storage
- Objects accumulated in memory buffer, written out to disk in large chunks
- Objects "recycled" to the front of the disk write position if frequently accessed, implementing an LRU

Storage: shortcomings

- No support for sparse objects
- No support for updating headers
  - Henrik’s storeUpdate rewrites the whole object out with fresh headers..
- Entire index is global and in memory!
- IO sizes are all wrong
- Lots of extra copying where its not needed

Storage: shortcomings

- Swap meta data log is written using sync disk operations
  - It wasn't written sync back in Squid-2.2..
- COSS is fast but still experimental and lacking in useful things like fast rebuild
- No useful way to distribute objects across disks "properly"
  - .. 10+ years of research into this one area!

Operating Systems

- Operating Systems implement buffer caches / VMs completely differently
- Squid's AUFS tuning was done for Linux-2.2, Solaris 2.6/2.7
- May not be applicable for Linux-2.6, FreeBSD, Solaris 10.

Operating Systems (ctd)

- Differences in handling disk writes
- Linux - seems to want to eat as much as it can during write() and flush it all out async
- FreeBSD/Solaris - write() may block depending upon filesystem semantics, not guaranteed to be async even with free buffer RAM
- == any sync write() is potentially FAIL!

Logfile Daemon

- Logfile Daemon - an example
  - Pipe logfile contents through local socket to external process
  - External process does the blocking writes
  - Squid tosses logfile data if buffering grows beyond a fixed size (default 128 * 64k)

Logfile Daemon (ctd)

- Results are pretty shocking:
  - Without logfile daemon: access.log writing on FreeBSD-6/7 and Linux-2.6 top out at ~ 500 req/sec
  - With logfile daemon: access.log writing exceeds 5000 req/sec under specific conditions
- .. but do the math.

Logfile Daemon (ctd)

- The math:
  - 500 req/sec
  - say, 80 char/req
  - Thats **40kbytes/sec** being written
- .. so obviously we're not filling buffer cache quickly with our 40kbytes/sec of logs; whats going on?

Logfile Daemon (ctd)

- Exposes underlying VM / buffer cache / filesystem operation
- We can't assume that disk writes will be sync at any point
- We can't assume OSes buffer things consistently
  - Linux vs FreeBSD raw device caching is different - important for COSS
  - Every screwup == drop in throughput

Disk IO patterns

- This may all change with flash based storage, still..
- .. disk IO is done in 4k chunks
- Disks generally handle >4k chunks about as fast as 4k chunks, up to about 64k
- (COSS at least reads in the whole object into memory and then returns 4k memory blocks as requested)
- Read/Write APIs must handle >4k ops!

Object Locality

- Objects are distributed two ways:
  - Round-robin
  - Least-load
- 10+ years of research shows "normal" web traffic includes temporal locality
  - Ie, fetching object generally implies subsequent fetching of other objects
- Would reduce disk IO substantially!

Object Locality

- Some reports that multiple COSS directories, more than a few (2? 3?) seem to not provide further speed improvements
- Objects from the same "page" are distributed across multiple storedirs
- Which means all storedirs have to get involved to fetch one page, instead of -one-storedir
- .. which is better? more or one storedir?

Object Locality

- More work is needed
  - .. ie, to bring Squid up to scratch with the other caching products out there.
- Luckily, this isn't new and unexplored territory - lots of research papers cover this stuff in quite a bit of detail.
  - .. some used Squid!

COSS: where to?

- COSS works great for small objects
- It doesn't intelligently work with the OS VM/ buffer cache at all
    - .. which is difficult to do cross-platform
- It doesn't handle rebuilds well
- It only stores objects with well-known sizes
    - .. a shortcoming in upper layers..

COSS: where to?

- A lot of stuff has been “tacked on” to fix flaws in the original design
- Original design: sync disk operations
- Adrian’s work: async disk operations, try to handle object relocation cases correctly
- Steven’s work: fix relocation logic to cut down on disk write IO and improve performance

COSS: where to?

- COSS could do with a rewrite
- Per-stripe metadata - improve rebuild times
- Storage API layer changes to allow for:
  - copy-free reads/writes, >4k sizes
  - store objects that can fit (ie, no Content-Length, but fully received)

New Storage Req's

- Lose the global index? Or support disk-only indexes?
- Support partial / sparse disk objects
- Separate out reply headers and body
  - To support header updates properly
  - .. and as part of a general code tidyup
- Threading/concurrency? Distributed stuff?
- "Shared" storage? (Eg NFS/shared FC FSes?)

Potential plan?

- .. well, a lot of this stuff doesn't need to live inside Squid at all.
- Think "memcached" but for disk objects
  - .. sort of a Google-like GFS?
- Simplifies development/testing; integration back into Squid may be difficult
- It -would- allow for interesting possibilities!
  - eg cheap SMP support, shared storage..

Potential plan?

1. Given what we know now, design a better API - not the best API, a “better” API
2. Implement some simple, naive memory/disk storage modules
3. Model/benchmark separate from Squid
4. At this point - we have more of an idea how to move forward!
   - instead of separate, small, incremental changes..

Questions?