Unbound in C

San Diego - 2006
Wouter Wijngaards
(wouter@NLnetLabs.nl)
Outline

- **Goals**
- **Design**
  - Server design
  - Module design
- **Major Issues**
  - Threads
  - Local zone server
  - Compression
- **Detail Issues**
  - Data Store
  - Spoofing Prevention
  - Overload Handling
Goals

- Validating recursive DNS resolver
- Another alternative open source implementation
- DNSSEC, RFC compliant, high performance
- Elegant design
- Portable C
- BSD License(?)
- NOT
  - an authoritative server
  - Feature bloat – difficult for a resolver
Server design options

• How to thread and do the workflow?
  – Looked into literature

• Event driven
  – `Select()` and events drive state machines
  – Every thread has all modules

• SEDA
  – Staged event driven arch
  – Queues to threadpools that do one module

• Discussion of these two options on next slides
• Positive
  – Queues reordered for cache
  – Unequal validation load could be moved

• Negative
  – Queues add enormous latency to requests
  – Queue and thread management problem
  – Slight downfall on DoS
  – Queue growth memory problem
Event driven

Main routine blocks in `select()` call
- Every module has a state, event-driven
- Process every request until finished or blocked.

Positive
- Good characteristics under heavy load
  - Requests are finished instead of queued up.
  - Less overhead in queuing, locks, thread scheduling

Negative
- Complicated due to stateful modules
- Validation load falls to thread that accepted request
Workflow

- Clean modules can be used for any design
- Modules to call another – from Unbound Java
Server design

- Server main puts requests in queue
- Handler
  - Look in msg cache
  - Calls modules
  - Send reply if done
- Messages from network can wake up a suspended request
Module Design – input!

State
- Per request
  - qname, type, class
  - Module state var
  - No buffers (plz!)

Per module
- Module caches
- Module config
- Module callbacks

Input
- Request
- Results from:
  - Module call
  - Network / timeout
  - Subrequest

module_activate()

Output
- Finished: result (ptr to msg)
- HandOver: Call next module
- Suspended (subreq, network)

Callbacks
- Custom alloc
- RRset cache
- Msg cache
- Network query
- Create subreq

Subreq to what module?
  - First, next, same

More callbacks?
Link and Compile

• Every module can be linked on its own against a main program
• **Main program provides callback services**
• Different main programs to make
  – Unit test programs
  – Resolver library
  – Remote (TCP) module connections
  – Server
• Valid, iter are clean modules but cache is still special.
Threading and forks

- **Threads**
  - Speed advantage on shared memory cache
  - As little locks as possible
  - Work without threads too

- **Every thread**
  - Listens on port 53
  - Listens to own port(s)
  - Own query list
  - Own local cache (called L1)

- **Shared - locked**
  - shared cache (called L2)
  - Request counts
  - malloc/free service
Caches – Need input!

- Caches
- RRset
- Msg-reply
- Trusted-key
- Infrastructure
- Where? L1 (local), L2 (shared).

- Clean cache design?
  - Generic L1-L2 fallback
  - Generic by datatype, module.
  - Some caches do
    - static config
    - Localzone serve
Local Cache

• L1: rbtree, hashtable.
  – LRU double linked list woven in, delete items to make room if at max size of the cache.
  – Timeout checked when access an item - refetch

Tree+LRU  Hashtable+LRU

http://www.nlnetlabs.nl/
Shared Caches

• L2: hashtable, locks per bucket.
  – Read: Copy data out – no locks per entry
    • Separate double linked LRU list?
      – Find an item to delete – snip off LRU list. Then delete in hashtable (get lock on buckets).
    • LRU updated on reads – how locking?
      – Unlock bucket, get lock on entire LRU list to update.
    • One big lock on LRU list. Bad. (input!)
Local zone server

- Need a local zone served (.localdomain)
- AS112 zones, do not leak
- **Unbound not authoritative server!**
- Options
  - NXDOMAIN (default for AS112)
  - Forward to (NSD) on host:port
- Basic service
  - No CNAME, DNAME, wildcards, NSEC ...
  - This is authoritative service!
  - Do it right or don't.
Compression

• Never uncompress incoming data:
  • Hard to store RRsets separately
• sendmsg/writev gather of uncompressed data
  • Use header, qname and rrset data without copying (!)
  • Have to update TTL values before send
  • Canonical rrset format ready for validation crypto
• copy&compress: use rbtree in L1 rrset cache for offsets
  • As a config option; copy=less cpu, compress=less bytes.
• Keep Rrsets locally compressed
  • Have to update compression ptrs and TTLs before send
  • Not canonical format
  • Imperfect compression ratio
Data store

• Packed RRset
  – Keeps wireformat RRset, ptrs to RRs, TTL.
  – Could keep RRSIG over the RRset as well
• TTL in absolute times
  – Use min TTL for RRsets, messages.
• Cache entries have validation status
• Store hashvalue in cache objects.
• dnames kept in wireformat, label offsets
• Ldns: No need to do all DNS constants again
Msg-RRset pointers

- `Msg(q+reply) consists of RRsets`
  - Keeping RRset inside msg is waste memory
  - `Rrset*`: hard to find/lock msg on rrset delete
- `First 64bits in RRset are creation ID.`
  - `thread_num` (16bit), `seq_number` (48bit).
  - `seq_number` wraps: clear cache / abort
- `Keep RRset* and ID, check ID on use.`
- **Reuse RRset memory only for RRsets**
  - Zero ID means RRset is not in use.
  - Copy RRset from/to cache gets new ID.
Spoof Prevention

• Random IDs:
  – Random() with initstate(256 bytes)

• port ranges:
  – Needed per thread (to listen easily)
  – Kqueue, kpoll() sys calls

• Scrubber for incoming messages
  – Routine in Iterator? Or Validator?
  – Spoofed NS additionalns confuse iterator
    • But get caught by validator afterwards
  – Scrubber as a module?
    • Between iterator and network.
Overload handling

• On overload answer from cache
• Detect overload
  – Request list is full
  – One thread: stop listen port 53
  – All threads: overload mode
    • Answer from cache or drop query.
• Schedule 1:2 ratio for port 53 : other ports
  – Does not depend on number of other ports
  – Drives towards completion of waiting queries
  – Every select: perform 0/1 port 53 and round robin the other ports handle at most 2.
Concept Module: Remote Cache module

A remote server
- Runs with a cache module only
- Store/Retrieve msg and reply
- Like remote msg cache
  - Localhost cache for nonthreaded pcs
  - For a resolver farm

Cache module
- Checks msg cache
- If not: network msg to cache server (suspend)
- If not: next module
- Result next module
  - Store on server
  - Finished(result).
Summary

• Event driven
• Modular design
  – Callbacks – minimal OO
  – Modules can call next module
  – Suspend waiting for network reply
• Threads: minimal, cache a copy
• Needs tweaks
  – Compression choice
  – Cache code
  – Module interfacing