Google

Profiling Ganeti 2.16

GanetiCon 2016, Dublin

Brian Foley (bpfoley@google.com)

This talk is pony-free!*

* Almost. Except for this slide. Friendship is not optimal! :-)

Ganeti @ Google:

- A fleet of large Ganeti clusters (see Ganeti @ Google talk for stats)
  - Clusters with up to 80 nodes
  - Xen/DRBD (=> daemons must fit in a small fixed-RAM dom0)
- Used Ganeti 2.10.x for a long time.
  - Stable, reliable, suboptimal in places, but well understood
- Wanted to upgrade to Ganeti 2.15/2.16
  - Needed for KVM/shared storage support
  - Wanted for shiny features like `gnt-filter`
  - Promise of scalability improvements (e.g. opportunistic locking)

Ganeti @ Google (2):

- Trial upgrade to 2.15 ~November 2015
  - Terrible performance on large clusters.
  - Many times slower than 2.10 with ordinary loads. Melt down with 'large' loads, e.g. queuing/running migrates for all insts on a node.
  - 100% CPU, swap pressure, unresponsive, lots of timeouts.
- Why?
  - masterd refactoring in Ganeti 2.12
  - Split multithreaded masterd into luxid, wconfd, per-job exec.py
  - Lots more RPC calls made, many copies of previously shared data

How to profile?

- Work is distributed between several daemons
  - confd, wconfd, luxid, noded, one exec.py per running job
  - CPU and memory use is transient -- no permanent leaks, and very spiky CPU use.
  - Hard to find why something is slow
    - Slow because of CPU/memory contention?
    - Being called far too much?
    - Simply a slow/inefficient implementation?
    - Contending on locks/other architectural bottlenecks?

How to profile? (2)

- "Apply standard sysadmin practices."
- Profile system load with atop/htop, iotop, pidstat, vmstat
  - All useful starting points
  - Gives aggregate statistics
    - What is the total RSS of all Ganeti daemons?
    - Is there a lot of disk IO? paging? CPU spikes?
  - But all these tools sample at ~1 Hz.
    - misses short lived processes
    - misses large temporary allocations
  - Black box => hard to pin down cause and effect

How to profile? (3)

- "Low-cost system-wide profiling of CPU/IO/syscalls with high frequency stack sampling? You're looking for Linux `perf`!"
- http://www.brendangregg.com/linuxperf.html!
- We'd *love* to use `perf`, but
  - `perf` relies heavily on stack walking and DWARF symbols
  - Haskell compiler didn't generate any DWARF until GHC 7.11
  - Python is interpreted, needs stack helpers to be useful
- However
  - perf execsnoop (trace all `execve()` calls) showed some wasteful `/sbin/lvs` usage from `ganeti-noded` we hadn't noticed before.

38c735d Using /sbin/lvs inefficiently

- `blockdev_getmirrorstatus_multi` noded RPC call does (approx.)

```
for disk in disks:
  output = _run_cmd(["/sbin/lvs", ..., disk.uuid])
  info.append(_parseLVInfo(output))
```

- Called during `gnt-cluster verify`
- `/sbin/lvs` takes ~50ms to run, so doesn't show up in *top.
- For DRBD is run twice per disk (once for _data LV, once for _meta LV)
- Is run for both primary and secondary disks on each node.
- So with 10 instances/node, this is 50ms * 10 * 4 = 2s on every node.
- Fix: `/sbin/lvs` can list all the LVs in one call, so let's do that!

I can't use `perf`, so What Do I Do Now?*

- Pick some easy workloads
  - `gnt-instance restart`
  - `gnt-cluster verify`
- Turn on debug logging (add `-d` to `/etc/defaults/ganeti`)
- Trace 'by hand' through the logs for a given job
  - `/var/log/ganeti/*.log`
- See if we can get a sense of the communication patterns, where the bottlenecks are, and what's expensive.

* Apologies to Louise Wener.

luxi and exec.py interactions

- luxi manages job queue/job status, and handles for QueryNodes etc:

  ```
  awk < luxi-daemon.log \
      '/New jobs enqueued: N/,/Finished jobs \(N,/{print}'
  ```

  (Beware background threads, interleaved work)
- `exec.py` runs the job: `grep ': job-N ' < jobs.log`
  - Makes RPC calls to wconfd, confd, luxid, noded
  - Writes JSON to `/var/lib/ganeti/queue/job-N` on state change.
  - luxi watchs `job-N` using inotify, re-reads updated state, and updates the in-memory state.
  - This happens every time a log message is appended.
  - This sounds slow -- $O(n^2)$ parse & serialize time for *n* messages.

d929e5b `ClusterVerifyConfig()` logs inefficiently

- `gnt-cluster verify` can generate lots of log messages
  - in verbose mode *or*
  - on a broken cluster *and*
  - is run periodically from gnt-watcher
- Instead of doing

```
for msg in self.VerifyConfig():
  self._ErrorIf(msg, ...) # Writes entirely new job-N file*
```

- Do it in batches
  - `VerifyConfig` returns a full msg list, so just update the job once.

* Then gzips, base64 encs, sends jobqueue_update RPC to all MCs (see `_AppendFeedback()`). Expensive!

c429dd2 luxi memory grows on job status updates

- luxi's job queue is represented using 3 lists

```
Queue { qEnqueued    :: [JobWithStat]
      , qRunning     :: [JobWithStat] -- lists of parsed JSON, file timestamps
      , qManipulated :: [JobWithStat]
      }
```

- luxi's inotify watcher updates the job queue when the files change
- But Haskell lists are immutable -- modifying creates a new copy...
  - `new_list = old_list.replace(old_jobstat, new_jobstat)`
- ...lazily
  - `new_list = (yield old_list.replace(old_jobstat, new_jobstat))`
- So if something doesn't read the list, a chain of 'thunks' (unevaluated functions) builds up with refs to jobstats. For us, ~1GB for 150 msgs!

Haskell 'space leaks'

- A common problem in Haskell programs
  - Make updates to an immutable data structure (eg list, Map, Set, IORef) without ever using the earlier version
  - Some time later the values are used/discarded.
  - In the meantime unevaluated thunks build up in memory.
  - Very easy to do by accident. Can also happen in other ways:

```
average :: [Float] -> Float
average [] = 0.0
average xs = sum xs / fromIntegral (length xs)
```

  - Usually fixed by adding a limited amount of strict evaluation.

Profiling Haskell space leaks

- Install profiling builds of all libraries (libghc-*-prof), build Ganeti with

```
./configure --enable-profiling \
  --haskell-flags="-O2 -caf-all --auto-all -rtsopts"
```

- Copy binaries onto test cluster
- Add `+RTS $RTSOPTS` to `/etc/default/ganeti` options

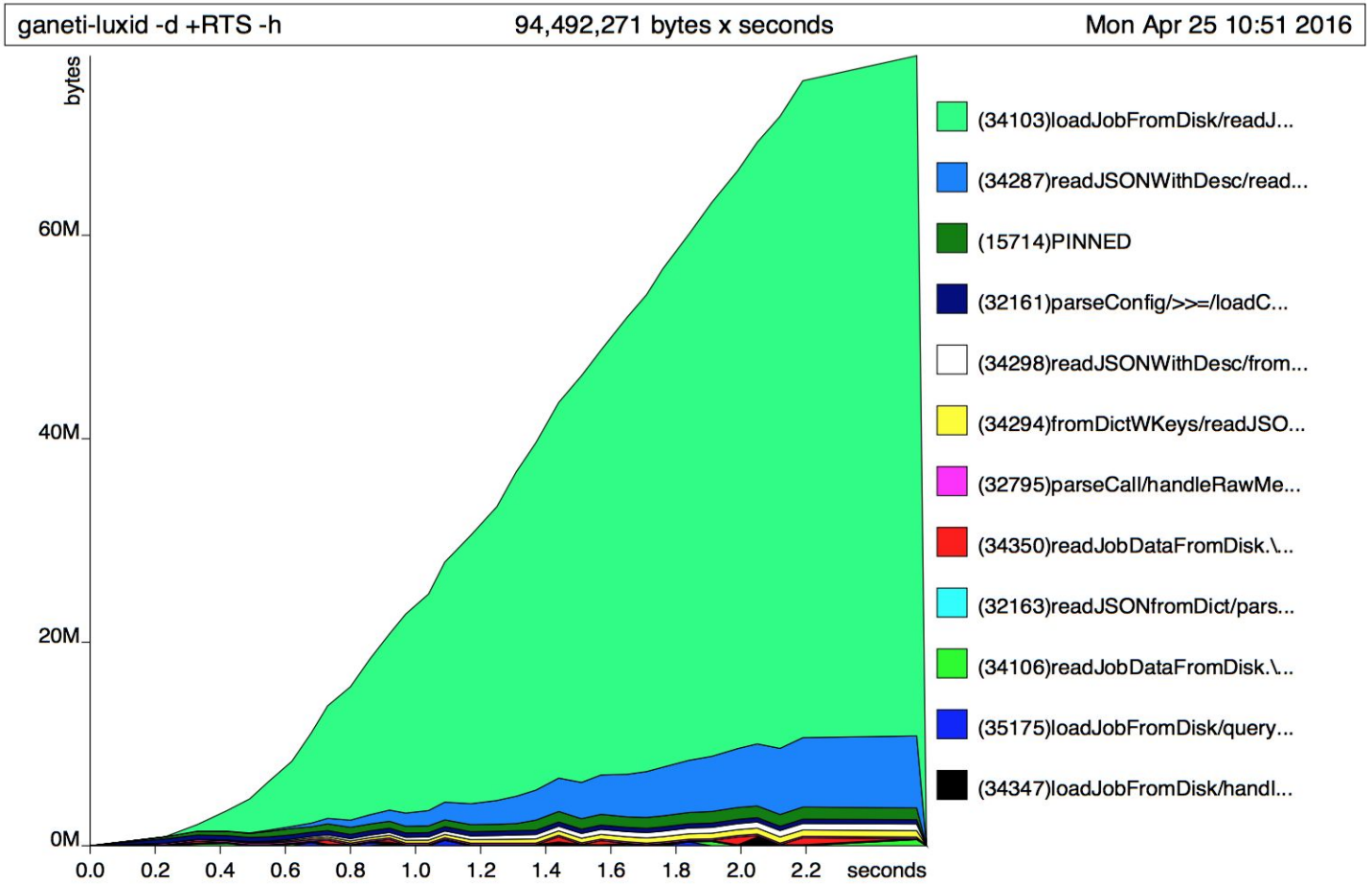
  -s for GC stats, -P for alloc/CPU profiles, -h for heap profiles,
- `chmod 777 /` ((!) so daemons can write profile files to their $PWD)
- Start Ganeti, run test workload, shut down Ganeti to dump profile.
- Graph `/ganeti-${FOO}.hp` with `hp2ps` (or `hp2any-manager`)
- Profiling is intrusive, inhibits optimisation, interpreting is a black art!
  - But lots of advice, e.g. Neil Mitchell & Rojemo & Runciman (96)

Allocation/CPU profile (-P)

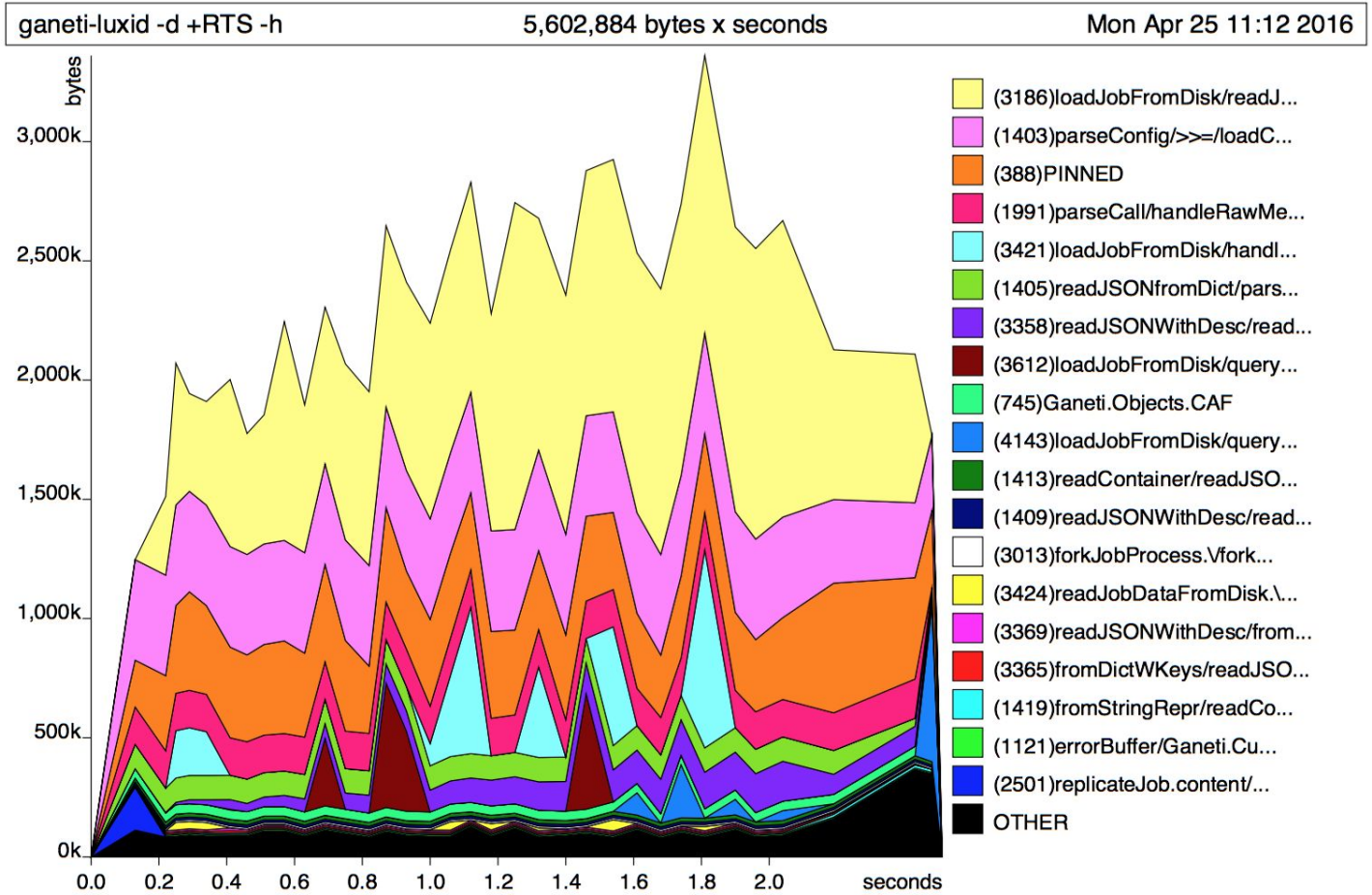
```
Time and Allocation Profiling Report  (Final)
         ganeti-luxid +RTS -P -RTS
       total time  =       49.71 secs   (49706 ticks @ 1000 us, 1 processor)
       total alloc = 8,618,876,280 bytes  (excludes profiling overheads)

COST CENTRE                    MODULE                     %time %alloc  ticks      bytes
parseConfig                    Ganeti.Config               22.8   35.2  11333 3035761344
curlMultiPerform.\             Ganeti.Curl.Multi            9.8    0.0   4870     464240
readJobDataFromDisk.\          Ganeti.JQueue                6.0    3.5   2958  301834040
readJSONfromDict               Ganeti.JSON                  5.4   12.7   2670 1091930016
readJSONfromDict.superfluous Ganeti.JSON                    4.3   11.5   2117  992242664
liftIO                         Ganeti.BasicTypes            4.3   14.3   2113 1231116640
fromDictWKeys                  Ganeti.OpCodes               3.6    2.2   1807  188409000
makeEasyHandle                 Ganeti.Curl.Multi            3.4    0.7   1695   58300408
logFormatter                   Ganeti.Logging               3.1    0.2   1539   15057504
withTimeout                    Ganeti.UDSServer             1.8    0.1    886    7749232
readJSONWithDesc               Ganeti.JSON                  1.8    0.5    876   45005688
attachWatcher                  Ganeti.JQScheduler           1.6    0.1    789   11577832
unstream/resize                Data.Text.Internal.Fusion    1.5    9.8    723  844336224
branchOnField                  Ganeti.JSON                  1.4    0.1    717    6211144
loadJobFromDisk                Ganeti.JQueue                1.4    2.7    678  230392264
maybeFromObj                   Ganeti.JSON                  1.1    0.4    565   34191656
logAt                          Ganeti.Logging               1.1    0.0    541    4066144
```

Before: Space leak (ignore absolute values)



After: Heap stable, much lower. Space leak fixed.



Test harnesses

- Ganeti profiles tend to have costs from lots of unrelated sources.
- Difficult to analyze, very difficult to reproduce without ext. interference.
- One strategy: extract specific functions into a small test harness.

```
main = do
  cfg <- loadConfig "config.data" >>= exitIfBad "Can't load config"
  n <- getNode cfg "node1.example.com"
  putStrLn $ nodeName n
  putStrLn "Before performMajorGc"
  performMajorGC
  putStrLn "Before exit"
  return ()
```

- Used to find f1574de Config parsing spends lots of time in getInstance

Test harnesses (2)

- Hard to extract luxi/wconfd RPC handlers into a test harness
  - Complex call stack & too much entangled state/state updates
- Another approach: Send individual 'probe' RPCs directly from a script.
- Ganeti RPC turns out to be easy to imitate* once you strip away all the obscuring code.
- This found/fixed 9825767 QueryInstances uses too much CPU/heap

```
cat <<EOF | ./query.py --server=luxid
{   "version":2150000,
    "method":"QueryNodes",
    "args":[[], ["sinst_cnt"], false]
} EOF
```

* Easy as in 50 lines of obvious straight-line Python that generates the same traffic as lib/rpc/client.py & co.

When all else fails, stare at the code until it confesses!

- cf077d3 shutdown instance pointlessly sleeps 5s for retry on success!
- 0a41418 cluster upgrade does 9 seq. ssh to each node (reduced to 6)
- 24da2b1 config replication does foreach mc { gzip; base64enc; send }
  (rather than gzip; base64enc; foreach mc { send })
- 9825767 RAPI-triggered getNodeInstance is very expensive
- 8ac8907 RAPI doesn't cancel jobs even if client times out
- My personal favourite:
- 46ac7ee WaitForJobChange holds onto old config until call returns
  - Jobs run from CLI without --submit
  - Jobs run from RAPI
  - Jobs watched with gnt-job watch

WaitForJobChange problem (heavily paraphrased)

- inotify thread:

```
while watch_config.wait_on_update():
    cfg = parse_config() # May update between thread forks. Big obj.
```

- listener thread:

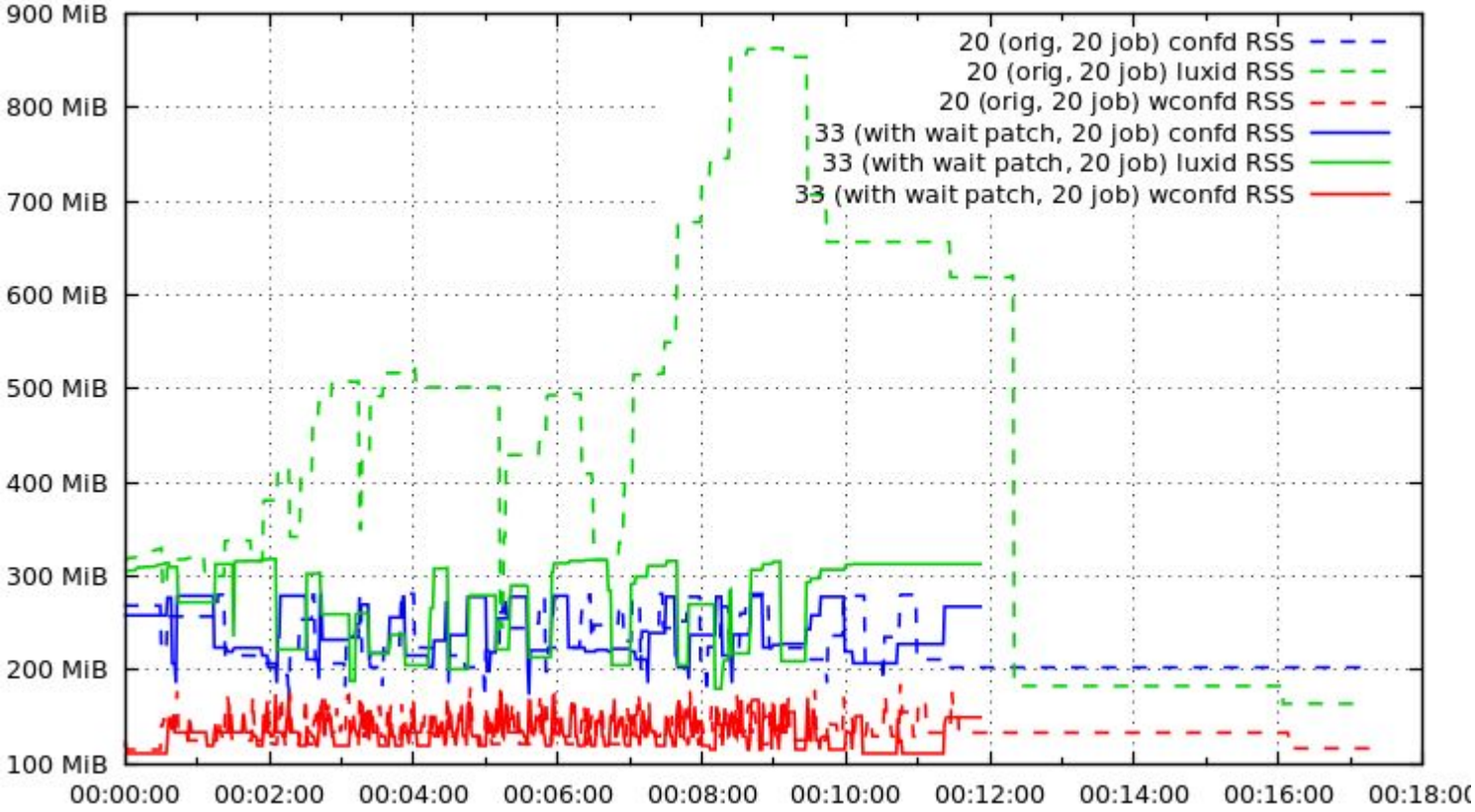
```
while True:
   conn = sock.accept()
   fork_hander_thread(conn)
```

- forked handler thread (H) (one per in-progress RPC call)

```
req = conn.recv()
(handler_func, args) = parse_req(req)
conn.send(handler_func(req, args, cfg))
```

- H refs `cfg` that was current when H began, until `send()`, even if unused

WaitForJobChange fix -- wconfd RSS stable



State of 2.15/2.16

- All these patches (+ several others) are on stable-2.15 HEAD
- In addition 2.16 has several pre-existing improvements over 2.15
  - Better buffering/cheaper sending of RPC responses.
  - More RPC calls to do cheaper in-place updates of config, instead of { lock/read from wconfd; modify in exec.py; write back to wconfd }
- Google have migrated *all* production Ganeti clusters to 2.16.
  - Really pleased with the performance. Prod-ready for our loads.
- Next:
  - Make a new patch release of 2.15. Get Debian/Ubuntu to pick up.
  - Release 2.16rc2, get tested by community. If OK, 2.16.0

(Unambitious) ideas for more scalability wins

- Job state update/watching is a heavyweight operation
  - We do it at least 4 times per job.
  - Make updateJob, appendJobLog, watchJobLog explicit RPC calls
    - (RPC by writing/watching shared files seems so 1970s!)
  - Make the log into a separate (text!) file and don't sync to MCs.
- exec.py expensively `recv()`s/parses entire config.data on startup
  - Extra work for luxi; uses per-job memory; mostly unneeded(?)
- config.data is huge, very verbose, and monolithic
  - Not (efficiently) machine readable, or (easily) human readable
  - Can we simplify disk data a bit? It's too complicated/redundant.
  - Maybe split disk.data / instance.data / config.data?

Thank you.

Questions?

Backup slide.

Opinionated, homeopathic amounts of evidence, and possibly entirely misguided.

More ambitious ideas for bigger scalability wins

- Replace Text.JSON with aeson in Haskell daemons. Death to String!
- Don't use JSON for bulk config/job RPC, use gRPC/Cap'n Proto/other
- Use a 'grown up' config distribution system, eg CoreOS's etcd. It has:
  - Lots of attention to heap use, speed, durability, availability
  - key/value store with hierarchical key names
  - Get/Update/Compare-and-Swap/Watch-subtree
  - transactions: IF (cond) { Update(k1,v1) } ELSE { Update(k2,v2) }
  - Paxos, cheap snapshots, rollback
  - stable 30k qps even during snapshots etc.
  - 'bigger' dependency, but mature, and Someone Else's Problem.
- Could easily store all of job state, lock state, cluster config.