while (true) do;

how hard can it be to keep running?

O'Reilly OSCON 2013 Portland, Oregon
Caskey L. Dickson/SRE Google, Inc.
caskey@google.com

Google

The (simple) goal

If you have a **non-trivial** number of servers
Keeping a given daemon running at all times on them is **hard**

Consider:

1 daemon x 300 servers x 6 months == 1 daemon x 1 server x 150 years

Google

Meet the foo service

- **eats resources** in a (usually) controlled fashion
- Has a **binary** (.../foo/bin/food) or three
- Runs as a specific user **account** "user foo"
- Has **static configuration** or resource data lib/foo/...
- Uses **command line flags** to manage the behavior
- Serves requests or processes data on a regular basis
- Internally packaged but **might be 3rd party** (OSS) upstream
- Periodic (controlled) **releases and updates** (weeks/months)

Google

Two halves of the problem

Getting the right thing running on the servers

- packaging
- configuration management
- upstream changes
- distribution
- installation
- upgrades
- rollbacks (not downgrades!)

Keeping it running that thing on the servers

- init
- upstart
- cron
- at
- bash
- health checkers

Google

KISS - What is the simplest possible solution?

1. Copy foo.tar.bz2 to every server
2. Unpack into /usr/local/foo
3. Add command-line to rc.local
4. Reboot server

(easy to implement using 'recipe' style management systems: e.g., ebuilds)

Pros

- Dead simple
- Easy turn-up
- Easy turn-down
- Per-host customization is “easy”*

Cons

- Manual everything
- No restart-on-crash
- No live-death detection

* ease of use is illusory–manual customization is neither scalable nor repeatable

Google

Packaged KISS - Add the package manager to the mix

1. Build foo.deb (or RPM, or ...)
2. post-inst scripts in foo.deb put init-scripts into place
3. Copy foo.deb to, and install on servers
4. Profit

Pros

- Leverages system package manager
- Easy turn-up
- Easy turn-down
- Marries upgrade to versions

Cons

- rollbacks can be tricky PM-fu
- Global rollouts
- Manual upgrades
- Implicit state
- Shared repo needed

Google

Private package repository with custom packages

(ignore the debian bias)

1. Set up a custom repo
2. Place your service's packages on it
3. Point your fleet at the appropriate version of the repo (testing, stable)
4. Put this in cron.d:

```
0 1,13 * * * root sleep $(( $RANDOM \% 21600 )); \
                  apt-get -qq update; \
                  apt-get -qq upgrade &>/dev/null
```

After provisioning a server:

```
$ apt-get install foo
```

Google

Packages as configuration management (an open debate)

Baking your configuration into packages has upsides and downsides. One-size-fits-all? One-size-fits-none?

Bare (distro style) packages:

- package contains only the software components, configuration files are pushed using puppet/cfengine/etc.
- introduces configuration/binary version sync problems

Staging packages:

- package with desired configs depends upon distro package
- remains 'dormant' or configured-off until activated via puppet or the like

Full packages:

- package installs and configures for running immediately
- scary, but powerful

Google

Package management can be solved (not the hard part)

Basic assumptions:

- Some means to designate a non-distro package to run on fleet (e.g., puppet, chef, cfengine, ...)
- Multiple host tracks (unstable, testing, stable)
- Package manager has pre- and post-install scripts that can be run as part of the package management
- package updates occur regularly (staggered) on fleet
- mechanism to assess current package versions fleet-wide
- OMGRCSBBQ!

Google

Init scripts - getting it running with init(-alike)

/etc/rc.local:

```
  /usr/bin/food -d
```

- Shared file, only use as a last-resort (i.e., never)

/etc/init.d/foo:

```
  case $1 in...
    start) food -d ;;
    stop) pkill food ;;
    restart) pkill food; food -d ;;
    status) ...? ;;
```

- Well known and supported
- Only really solves the start-at-boot problem, what if foo crashes?

Google

Init scripts–keeping it running with cron (KISS approved)

/etc/cron.d/foo:

```
*/5 * * * * test -x /etc/init.d/foo && \
                /etc/init.d/foo start
```

- Every five minutes, (try to) make it start
- Not terrible, better than nothing
- No way to admin-down a service, e.g., during upgrades
- Workarounds are error-prone (chmod -x, forget to +x later)
- What keeps cron running?
- atd has similar concerns
- restart storms - watch your startup dependencies!
- can trash pid files e.g., **concurrent starts**

Google

Init scripts–upstart is better, somewhat

/etc/init/foo.conf:

```
  start on runlevel [2345]
  stop on runlevel [!2345]
  expect fork
  respawn
  exec /usr/bin/food -d
```

- New(-ish) hotness
- Will respawn on process crash - bonus!
- Attempts to track forks DANGER
- Has an expect SIGSTOP option, interesting
- No way to contradict upstart's conception of process state (upstart thinks your job is down when up == pain)
- *Implicit* 10 restarts/5 seconds limit, then **gives up forever**!

Google

Flags: in init vs. launch-script

/etc/init/foo.conf:

```
  exec /usr/bin/food -d
```

- Updating flags requires new conf file
- Complicates dev-testing
- Only allows for simple startup sequences
- Robustness demands a separate 'make sane and launch' script

/etc/init/foo.conf:

```
  exec /usr/bin/run-foo
```

/usr/bin/run-foo:

```
  #!/bin/bash
  if [[ -e ... ]]; then ...
  mkdir -p /var/lib/foo/...
  /usr/bin/food -d
```

Google

Daemonization requires pidfiles (this is bad)

*"There are **no useful checks** that can be made with a pidfile." – me*

manipulating pid files

- read-modify-write cycle
- no locking facilities are used to assure correctness
- only valid if all handling is correct *at all times*

Fragile System: any system whose correct behavior relies upon all actors perfect behavior at all times.

```
$ /etc/init.d/foo start # process backrounds itself
```

did food launch? is foo still running? How to check? What do I kill to stop foo?

Google

Daemonization and why it is a bad idea (under upstart)

"Is foo running still, does it need to be launched again?"

/etc/init/foo.conf:

```
  expect fork
  respawn
  exec /usr/bin/food -d
```

This seems like a good idea, so long as food is a C/C++ program that **precisely implements daemonization using ONLY the single or double-fork method** (read: Fragile System).

What if we later add setup/maintenance/cleanup steps?

Google

Daemonization and why it is a bad idea (under upstart)

/etc/init/foo.conf:

```
  expect fork  # DANGER!
  respawn
  exec /usr/bin/run-foo
```

/usr/bin/run-foo:

```
  #!/bin/bash
  # First we do some sanity cleanups on system
  if [[ -e ... ]]; then ...
  mkdir -p /var/lib/foo/...
  # launch foo
  /usr/bin/food -d
```

Daemonization breaks the only reliable control link available–process ownership/parentage. We **NEED** to get a signal when the process dies. Reliably. No, really, reliably.

Google

Correct process control under upstart (STEP 1)

/etc/init/foo.conf:

```
  respawn
  exec /usr/bin/run-foo
```

/usr/bin/run-foo:

```
  #!/bin/bash
  # First we do some sanity cleanups on system
  if [[ -e ... ]]; then ...
  mkdir -p /var/lib/foo/...
  # launch foo
  exec /usr/bin/food --foreground
```

SAFE: process run-foo becomes the daemon, upstart retains control over the actual process which is the daemon, forks done to prepare environment are irrelevant. **May be arbitrarily nested.**

Google

Correct process control under upstart (STEP 2)

Upstart will eventually give up launching your program if you let it.

Therefore we still need cron to tell upstart otherwise.

/etc/cron.d/foo:

```
*/5 * * * * test -r /etc/init/foo.conf && \
              initctl start foo
```

Now we are (finally) ahead of the game.

1. Upstart will continually try to launch our program.
2. Cron operation will be a NOOP if upstart still controls the child it forked.
3. If upstart gives up, cron will tell it to get back to work.

The method by which you start, stop and restart your daemon is an interface with a contract.

Be sure you understand the semantics of that contract. Especially as it affects upgrades and rollbacks.

Google

The contract: *pkill -u ${role} food*

To admin-stop your job:

1. chmod -r /etc/init/foo.conf
2. initctl stop foo
3. pkill -u ${role} food

Steps 1 of stop/start are done during package removal/upgrade for us.

To start your job:

1. chmod +r /etc/init/foo.conf
2. wait for cron *or*
3. initctl start foo

To restart your job:

1. pkill -u ${role} food
2. wait for upstart to relaunch

The contract: run-foo

- run-foo is a script that launches foo
- It non-destructively prepares the runtime environment
- makes no assumptions as to prior state

  start-time is recovery-time
- *If the environment is sane*, it execs the actual daemon in

  foreground mode
- If not sane, log to stdout, **sleep for a while**, then exit abnormally
- Look to /var/log/upstart/foo.log for debugging

Google

upgrade flow in general (remove the old package)

1. old-package **pre-rm** script
   - ideally empty/nonexistent
   - beware of unexpected restarts
   - no split-operations
2. old package files removed
   - magically prevents automatic restarts at this point forward
3. old-package **post-rm** script
   - (try to) shut down this version by killing it
   - asynchronous commands only
   - beware of deadlocks and hangs–wait with care
   - no split-operations

Google

upgrade flow in general (install the new package)

1. new-package **pre-inst** script
   - don't assume the pre/post rm scripts of the last package worked
   - don't assume the previous version of the package was installed
   - no split-operations
   - kill existing versions just in case
2. new-package files emplaced
   - you may get automatically started at any time without knowing it (sanity checks help)
3. new-package **post-inst** script
   - make sure the old version is gone/force a restart
   - no split operations

Google

Health checks
pkill running job if check fails or times out

Periodic ACTIVE tests to verify service is responding in *any* way.

- simple as port-connect + banner checks

```
nc localhost 22 < /dev/null > /dev/null \
  && echo yep || echo nope
```

- complex as deep black-box end-to-end tests

```
curl -s localhost > /dev/null \
  && echo yep || echo nope
```

Not acceptable (no process interaction):

- pgrep
- kill -0
- netstat
- pidfile checks

Google

Issues that must be solved
(and how we've done it)

- Regular updates (cron updates)
- Safe rollbacks, rollback != downgrade (cron updates)
- Changes to daemon flags (run-foo)
- Changes to static configs (pkill on upgrade)
- Changes to the binary (pkill on upgrade)
- Version compatibility (combined configs + bins)
- Restart on upgrade (upstart respawn)
- Restart on exit (upstart respawn)
- Restart on live death (health checks + pkill)
- Liveness tests (health checks)
- Quarantine (remove package, or /etc/init/foo.conf)

Google

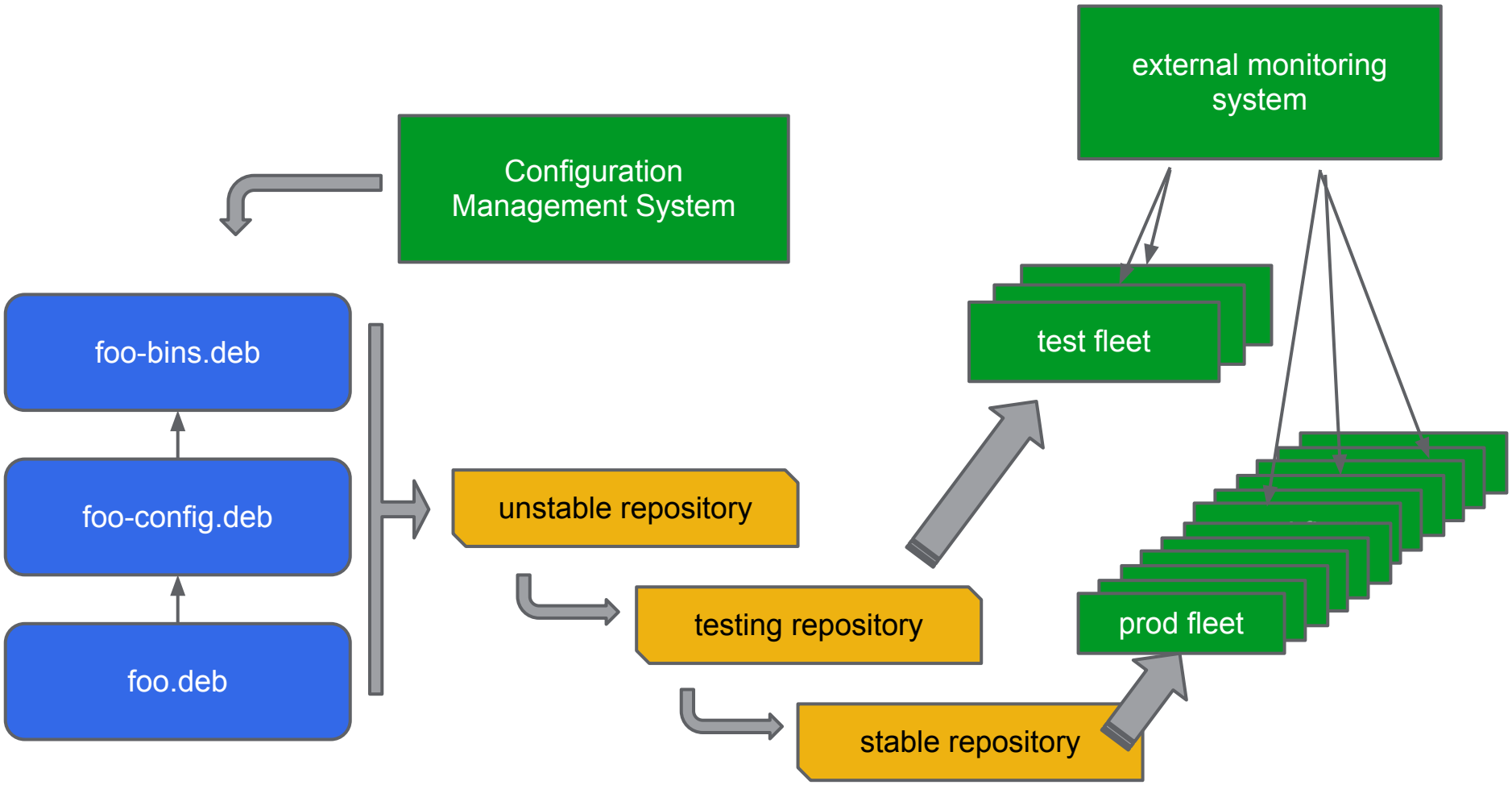
Landmines

- configuration management
- rollbacks, never downgrades
- hung package manager
- ineffective fleet state information
- hung daemons (insufficient health checks)
- race conditions in scripts
- virtualization overcommit

Google

The recipe (finally)

- All state in **revision control** system--build packages from head
- Complete package(s), **bins + configs**
- **Multiple repos** (unstable/testing/stable)
- Server fleet assigned to specific repos
- Test fleet with **real traffic**
- **Automatic installation** of promoted packages
- **Don't daemonize** or use pidfiles (race conditions)
- Start time is recovery time (**constant death**)
- Reliable upgrades (**pkill**)
- Local **monitoring** (upstart + cron/nc/wget)
- Remote **monitoring** (your choice)
- Kill to **gain control**


external monitoring system
Configuration Management System
test fleet
foo-bins.deb
foo-config.deb
foo.deb
unstable repository
testing repository
stable repository
prod fleet

The End

feedback welcome
caskey@google.com

join us
google.com/jobs

Extra Material

the case against pidfiles

Google

What are pidfiles good for?
Status: No pid file

Possible states:

- Daemon not running
- Daemon exited cleanly
- Daemon running but PID file removed
  - errant shutdown script
  - concurrent start of daemon

Action to take (desired state = running):

- assume down
- **attempt start** of daemon
  (rely upon daemon to not doubly-run)

Action to take (desired state = stopped):

- hope it is down
- or **attempt kill** of daemon

Google

What are pidfiles good for?

Status: Pid file with invalid process id (kill -0 fails):

Possible states:

- Daemon not running
- Daemon exited w/o pid file cleanup
- Daemon running under different PID
  - PID file overwritten by subsequent start attempt
  - PID file creation failed
  - concurrent start of daemon

Action to take (desired state = running):

- assume down
- **attempt start** of daemon
  (rely upon daemon to not doubly-run)

Action to take (desired state = stopped):

- hope it is down
- or **attempt kill** of daemon

Google

What are pidfiles good for?

Status: Pid file with valid process id (kill -0 passes)

Possible states:

- Daemon not running (stale pid matches other process)
- Daemon exited w/o pid file cleanup (again stale pid match)
- Daemon running under different PID (stale pid match)
  - PID file overwritten by subsequent start attempt
  - PID file creation failed
- Daemon actually running

Action to take (desired state = running):

- hope it is up
- or **attempt start** of daemon just in case
  (rely upon daemon to not doubly-run)

Action to take (desired state = stopped):

- assume pid is the daemon
- **attempt kill** of that process

Google

What are pidfiles good for?

Status: **ANY PIDFILE STATE**

Pidfiles have no practical use in large scale system administration. The give no actionable data to automated management scripts. Code to create/maintain them or that relies on them is merely a source of bugs rife with **race conditions**.

Action to take (**desired state = running**):

- or **attempt start** of daemon just in case (rely upon daemon to not doubly-run)

Action to take (**desired state = stopped**):

- **attempt kill** of that process