Lightweight Containerization at Facebook

Zoltan Puskas (zpuskas@fb.com)
Production Engineer on Infrastructure
Agenda

• What is Tupperware?
• Why use Btrfs?
• Building layered images
• Launching with systemd
• Results
**Terminology**

*Infrastructure we run on*

- Data Center: One or more clusters
- Cluster: multiple racks
- Rack: multiple machines
- Tupperware Job: a collection of similar tasks
- Task: instance of a binary running
Tupperware

Our solution

• Tupperware is a massively parallel job execution framework
• Runs a many of Facebook’s services
• Isolated runtime environment
• Resource controlled
• Easy workflow
Tupperware

But why?

• Why not Docker/CoreOS:
  • They did not exist when Tupperware started
  • Integration with Facebook systems

• Why not VM?
  • Performance penalty
  • Harder to debug issues
## Architecture

### Container Stack

<table>
<thead>
<tr>
<th>Industry</th>
<th>Facebook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etcd, Consul</td>
<td>Zookeeper based discovery service</td>
</tr>
<tr>
<td>Kubernetes, Docker Swarm, Chronos</td>
<td>Tupperware Scheduler</td>
</tr>
<tr>
<td>Docker Networking, CoreOS Flannel</td>
<td>Tupperware ILA</td>
</tr>
<tr>
<td>Docker Engine, RKT</td>
<td>Tupperware Agent</td>
</tr>
<tr>
<td>KVM, Hyper-V, LXC</td>
<td>Facebook Hosts</td>
</tr>
</tbody>
</table>
## Architecture

### Building Blocks

<table>
<thead>
<tr>
<th>Industry</th>
<th>Facebook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docker CLI, kubectl</td>
<td>Tupperware CLI</td>
</tr>
<tr>
<td>Chef, Puppet, Ansible</td>
<td>Job Spec, Chef</td>
</tr>
<tr>
<td>Docker registry, CoreOS registry</td>
<td>Distributed package manager</td>
</tr>
<tr>
<td>Dockerfile and image, CoreOS image</td>
<td>Buck target and Btrfs Image</td>
</tr>
</tbody>
</table>
Architecture

Overview

- job_config.tw
- Scheduler (job)
  - CLI
  - Host (tasks)
  - Host (tasks)
  - Host (tasks)
  - Server DB
  - Resource Manager
- Package storage system
Architecture

Tupperware Agent

Agent
- Task manager
- Package manager
- Volume manager
- Resource manager
- Scheduler heartbeat

Agent-helper
- task_1
- task_2
- task_3
Launching containers

Moving to images

• Previous solution
  • rootfs.tar.gz
  • Extract a copy per task
  • yum install during prepare
  • busybox init via LXC
  • cgroup v1

• New solution
  • Self contained Btrfs image
  • RW snapshots on RO base
  • Everything is pre-installed
  • Systemd init via nspawn
  • cgroup v2
Lightweight containers

What do we want from our containers?

• Reliability
• Determinism
• Transparency
• Low overhead
• Easy end to end workflow
Image Layering
A better way to manage runtimes

- Running Task
- Application Image
- Facebook Image
- Base OS Image
Btrfs

B-trees everywhere

- Copy-On-Write
- Subvolumes
- Snapshots (RO and RW)
- Binary diffs (send/receive)
- Quotas
- Cgroups IO control
Btrfs Subvolumes

```
vol_1  ->  /data/vol_1
vol_2  ->  /data/vol_2
vol_3  ->  /data/vol_3
```
Subvolumes

Creating a Btrfs filesystem

```bash
# truncate -s 1G image.btrfs
# mkfs.btrfs image.btrfs
...
Filesystem size: 1.00GiB
Block group profiles:
  Data:    single        8.00MiB
  Metadata: DUP          51.19MiB
  System:   DUP           8.00MiB
...
# mkdir root
# mount image.btrfs data
# mount
...
/tmp/demo/image.btrfs on /tmp/demo/data type btrfs (rw,relatime,space_cache,subvolid=5,subvol=/)```
# btrfs subvolume create data/vol_1
# btrfs subvolume create data/vol_2
# btrfs subvolume create data/vol_1
# ls data
vol_1  vol_2  vol_3
# touch data/vol_1/foo
# touch data/vol_2/bar
# ls data/vol_1/
foo
# ls data/vol_2/
bar
# ls data/vol_3/
# btrfs subvol list data/
ID 259 gen 17 top level 5 path vol_1
ID 260 gen 18 top level 5 path vol_2
ID 261 gen 18 top level 5 path vol_3
# mkdir vol_1_separate && mount -o subvol=vol_1 image.btrfs vol_1_separate
# ls vol_1_separate
foo

Subvolumes
Basic subvolumes
Btrfs snapshots

- vol_1
  - vol_1_2
    - vol_1_2_3

- /data/vol_1
- /data/vol_1_2
- /data/vol_1_2_3
Subvolumes
Snapshots and nesting

# btrfs subvolume snapshot data/vol_1 data/vol_1_2
# ls data/vol_1/
foo
# ls data/vol_1_2/
foo
# touch data/vol_1_2/baz
# ls data/vol_1_2/
foo baz
Copy on write and diffs

Building a tree of image layers

- Lower disk space usage
- Lower disk IO usage
- Improved disk data caching
- We can independently version layers
- Different update schedules for layers
Copy-on-Write with Diffs
Restore layers with diffs
<table>
<thead>
<tr>
<th>Task</th>
<th>App1</th>
<th>Task</th>
<th>App2</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>📦</td>
<td>✓</td>
<td>📦</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>🌞</td>
<td></td>
<td>🌞</td>
<td></td>
</tr>
<tr>
<td></td>
<td>💻</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Base OS**

**SSH, Cert, ...**
Building images

Buck build

- Declarative image building
- Fast parallel builds
- Reproducible builds
- Incremental builds
- Separation of build and run time
Building images

Buck build

• Fully self contained
• Provides true FS level isolation
• Build once, use everywhere
• Testable
Building images

Custom image build
Image buck TARGET
Define your image!

image_layer(
  name='job_layer',
  parent_layer=':minimal_busybox',
  copy_deps=[
    (':hello_world.tar', '/foo/bar/'),
    {
      'source': ':hello_world.tar',
      'dest': '/foo/bar/hello_world_again.tar',
      'group': 'nobody',
    },
    (':cool_service', '/usr/bin/cool_service'),
  ],
  features=[':some_feature_dirs_and_rpms'],
)

java_library(
  name = 'collect',
  srcs = glob(['*.java']),
  deps = [
    '//java/com/facebook/base:base',
  ],
)

java_binary(
  name = 'cool_service',
  deps = [
    ':collect',
    ':base',
  ],
)
Image buck TARGET
Define your image!

image_layer(
    name='miminal_busybox',
    tarballs=[
        ('//tools/build/buck/container_images:busybox.tgz', '/'),
    ],
    features=[':some_feature_dirs_and_rpms'],
)

image_feature(
    name='some_feature_dirs_and_rpms',
    rpms={
        'jdk_8u60': 'install',
    },
    make_dirs=[
        '/foo/bar',
        ('/foo/bar', 'baz'),
        {'path_to_make': 'borf/beep',
         'into_dir': '/foo',
         'user': 'nobody',
        },
    ],
    features=[],
)
Control volume

Operating on layers

```
/  
   |-- images
   |   |-- OS.base
   |   |-- OS.base.jdk
   |   
   |-- tasks
   |   |-- task_1
   |   
   |-- tmp
   
   `-- taks_2
```
**systemd**

Running the container, running inside the container

- systemd-nspawn
- systemd-init
- Container aware
- Logging to outside of the container (too)
- Running the container at build time
Performance

It’s fast
Performance

Low IO
Performance

Prefetch magic
Performance

Still low IO
Fast iteration

Develop, test and deploy

• Tools that enter, run and debug images
• Automated image build system with dependency handling
• Distribution via existing package management system
• Automated releases
Summa summarum

Building better containers

- Tupperware helps us scale Facebook
- Using a next generation file system
- Flexible images
- Standard containerization tools
- Higher efficiency with more robustness
Questions?

Zoltan Puskas (zpuskas@fb.com)