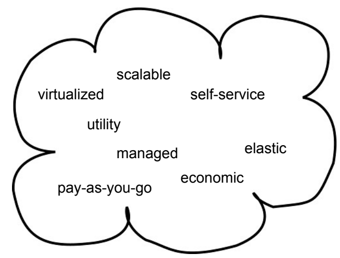
(Private) Cloud Computing with Mesos at Twitter

Benjamin Hindman
@benh

what is cloud computing?



what is cloud computing?

- "cloud" refers to large Internet services running on 10,000s of machines (Amazon, Google, Microsoft, etc)
- "cloud computing" refers to services by these companies that let <u>external customers</u> rent cycles and storage
  - Amazon EC2: virtual machines at 8.5¢/hour, billed hourly
  - Amazon S3: storage at 15¢/GB/month
  - Google AppEngine: free up to a certain quota
  - Windows Azure: higher-level than EC2, applications use API

what is cloud computing?

- cheap nodes, commodity networking
- self-service (use personal credit card) and pay-as-you-go
- virtualization
  - from co-location, to hosting providers running the web server, the database, etc and having you just FTP your files … now you do all that yourself again!
- economic incentives
  - provider: sell unused resources
  - customer: no upfront capital costs building data

"cloud computing"

- infinite scale …

From: [redacted]
To: [redacted]
Cc: [redacted] Benjamin Hindman <benh@EECS.Berkeley.EDU>; [redacted]
Sent: Wed May 05 12:31:24 2010
Subject: Re: Question on recent AWS usage

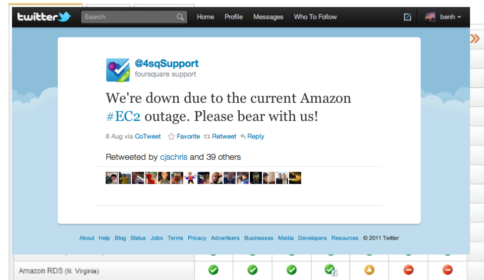
Hi [redacted]

Hope things are well with you. I'm not sure if anybody from the RAD Lab has been in touch with you about this, but a big paper deadline is coming up and several projects in the RAD Lab are using EC2 extensively for research experiments and we are hitting our limit. The deadline is Friday and I'm wondering if we can get the limit increased temporarily until Friday. I think our limit may currently be 500 instances, could we get it increased to a 1000 or 2000?

[redacted]
CS Graduate Student
UC Berkeley

"cloud computing"

- always available …



challenges in the cloud environment

- cheap nodes fail, especially when you have many
  - mean time between failures for 1 node = 3 years
  - mean time between failures for 1000 nodes = 1 day
  - **solution:** new programming models (especially those where you can efficiently "build-in" fault-tolerance)
- commodity network = low bandwidth
  - **solution:** push computation to the data

moving target

infrastructure as a service (virtual machines)
  ➔ software/platforms as a service

why?

- programming with failures is hard
- managing lots of machines is hard

moving target

infrastructure as a service (virtual machines)
➔ software/platforms as a service

why?
- **programming with failures is hard**
- **managing lots of machines is hard**

programming with failures is hard

- analogy: concurrency/parallelism
  - imagine programming with threads that randomly stop executing
  - can you reliably detect and differentiate failures?
- analogy: synchronization
  - imagine programming where communicating between threads might fail (or worse, take a very long time)
  - how might you change your code?

problem: distributed systems are hard

solution: abstractions (higher-level frameworks)

MapReduce

- Restricted data-parallel <u>programming model</u> for clusters (automatic fault-tolerance)
- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by Apache Hadoop project
  - Used by Yahoo!, Facebook, Twitter, …

beyond MapReduce

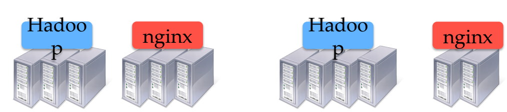
- many other frameworks follow MapReduce's example of restricting the programming model for efficient execution on clusters
  - **Dryad** (Microsoft): general DAG of tasks
  - **Pregel** (Google): bulk synchronous processing
  - **Percolator** (Google): incremental computation
  - **S4** (Yahoo!): streaming computation
  - **Piccolo** (NYU): shared in-memory state
  - **DryadLINQ** (Microsoft): language integration
  - **Spark** (Berkeley): resilient distributed datasets

everything else

- web servers (apache, nginx, etc)
- application servers (rails)
- databases and key-value stores (mysql, cassandra)
- caches (memcached)
- all our own twitter specific services …

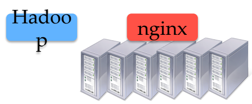
managing lots of machines is hard

- getting efficient use of out a machine is non-trivial (even if you're using virtual machines, you still want to get as much performance as possible)



managing lots of machines is hard

- getting efficient use of out a machine is non-trivial (even if you’ re using virtual machines, you still want to get as much performance as possible)

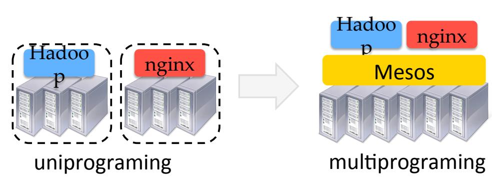


problem:
lots of frameworks and services
… how should we allocate resources (i.e., parts of a machine) to each?

idea:
can we treat the datacenter as one big computer and **multiplex** applications and services across available machine resources?

solution: mesos

- common resource sharing layer
  - abstracts resources for frameworks



twitter and the cloud

- owns private datacenters (not a consumer)
  - commodity machines, commodity networks
- not selling excess capacity to third parties (not a provider)
- has lots of services (especially new ones)
- has lots of programmers
- wants to reduce CAPEX and OPEX

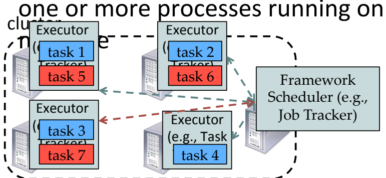
twitter and mesos

- use mesos to get cloud like properties from datacenter (private cloud) to enable "self-service" for engineers

(but without virtual machines)

computation model: frameworks

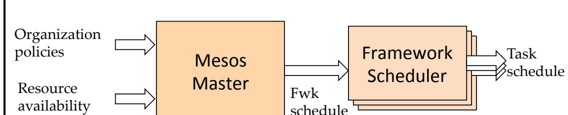
- A ***framework*** (e.g., Hadoop, MPI) manages one or more ***jobs*** in a computer cluster
- A ***job*** consists of one or more ***tasks***
- A ***task*** (e.g., map, reduce) is implemented by one or more processes running on a single



Job 1: tasks 1, 2, 3, 4
Job 2: tasks 5, 6, 7

23

two-level scheduling



- Advantages:
  - Simple → easier to scale and make resilient
  - Easy to port existing frameworks, support new ones
- Disadvantages:
  - Distributed scheduling decision → not optimal

24

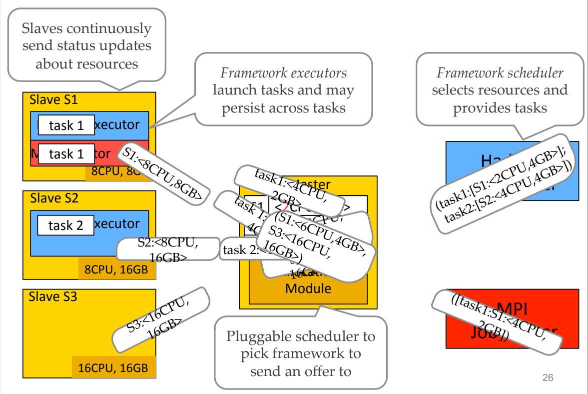
resource offers

- Unit of allocation: ***resource offer***
  - Vector of available resources on a node
  - E.g., node1: <1CPU, 1GB>, node2: <4CPU, 16GB>
- Master sends resource offers to frameworks
- Frameworks select which offers to accept and which tasks to run

Push task scheduling to frameworks

25

Mesos Architecture: Example



26

twitter applications/services

if you build it …
they will come

let's build a url shortner (t.co)!

development lifecycle

1. gather requirements
2. write a bullet-proof service (server)
   - load test
   - capacity plan
   - allocate & configure machines
   - package artifacts
   - write deploy scripts
   - setup monitoring
   - other boring stuff (e.g., sarbanes-oxley)
3. resume reading timeline (waiting for machines to get allocated)

development lifecycle with mesos

1. gather requirements
2. write a bullet-proof service (server)
   - load test
   - ~~capacity plan~~
   - ~~allocate & configure machines~~
   - package artifacts
   - write ~~deploy~~ configuration scripts
   - ~~setup monitoring~~
   - other boring stuff (e.g., sarbanes-oxley)
3. ~~resume reading timeline~~

t.co

- launch on mesos!

CRUD via command line:

```
$ scheduler create t_co t_co.mesos
Creating job t_co
OK (4 tasks pending for job t_co)
```

t.co

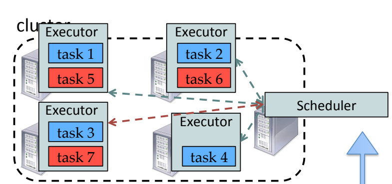
- launch on mesos!

CRUD via command line:

```
$ scheduler create t_co t_co.mesos
Creating job t_co
OK (4 tasks pending for job t_co)
```

tasks represent shards

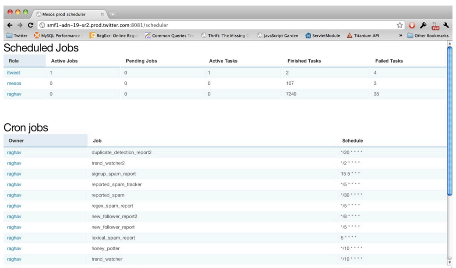
t.co




```
$ scheduler create t_co t_co.mesos
```

t.co

- is it running? (“top” via a browser)



Mesos prod scheduler

smf1-adn-19-sr2.prod.twitter.com:8081/scheduler

Twitter | MySQL Performance | RegExr: Online Regul | Common Queries Tre | Thrift: The Missing G | JavaScript Garden | ServletModule | Titanium API | Other Bookmarks

Scheduled Jobs

Role	Active Jobs	Pending Jobs	Active Tasks	Finished Tasks	Failed Tasks
itweet	1	0	1	2	4
mesos	0	0	0	107	3
raghav	0	0	0	7249	35

Cron jobs

Owner	Job	Schedule
raghav	duplicate_detection_report2	*/20 * * * *
raghav	trend_watcher2	*/2 * * * *
raghav	signup_spam_report	15 5 * * *
raghav	reported_spam_tracker	*/5 * * * *
raghav	reported_spam	*/30 * * * *
raghav	regex_spam_report	*/5 * * * *
raghav	new_follower_report2	*/8 * * * *
raghav	new_follower_report	*/5 * * * *
raghav	lexical_spam_report	5 * * * *
raghav	honey_potter	*/10 * * * *
raghav	trend_watcher	*/10 * * * *
raghav	duplicate_detection_report	05 */2 * * *

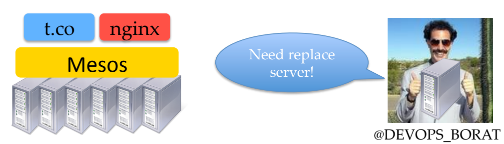
what it means for devs?

- write your service to be run anywhere in the cluster
- anticipate ‘kill -9’
- treat local disk like /tmp

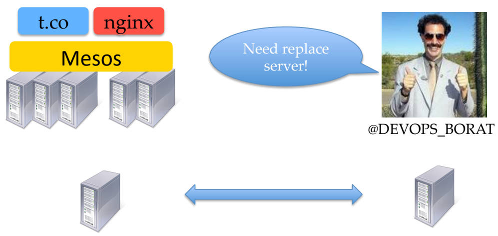
bad practices avoided

- machines fail; force programmers to focus on shared-nothing (stateless) service *shards* and *clusters*, not machines
  - hard-coded machine names (IPs) considered harmful
  - manually installed packages/files considered harmful
  - using the local filesystem for persistent data considered harmful

level of indirection #ftw



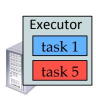
level of indirection #ftw



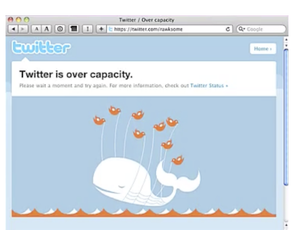
level of indirection #ftw

example from operating systems?

isolation

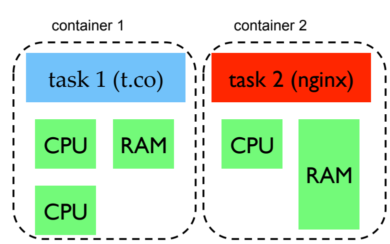


what happens when task 5 executes: `while (true) {}`



isolation

- leverage linux kernel containers



software dependencies

1. package everything into a single artifact
2. download it when you run your task

(might be a bit expensive for some services, working on next generation solution)

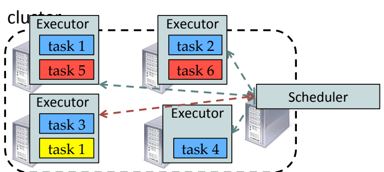
t.co + malware

what if a user clicks a link that takes them some place bad?

let's check for malware!

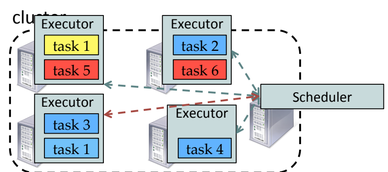
t.co + malware

- a malware service already exists … but how do we use it?



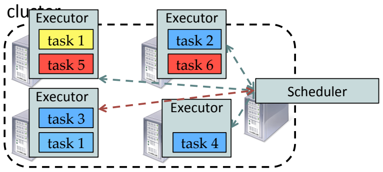
t.co + malware

- a malware service already exists … but how do we use it?



t.co + malware

- a malware service already exists … but how do we use it?



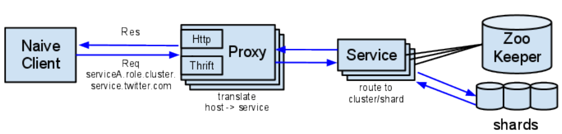
how do we name the malware service?

naming part 1

- service discovery via ZooKeeper
  - zookeeper.apache.org
- servers register, clients discover
- we have a Java library for this
  - twitter.github.com/commons

naming part 2

- naïve clients via proxy



naming

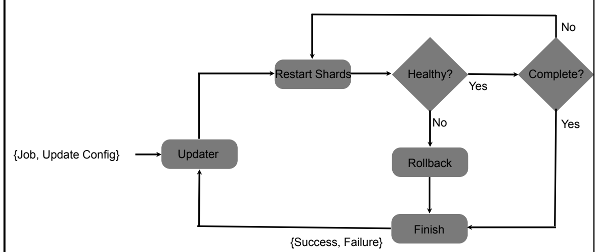
- PIDs
- /var/local/myapp/pid

t.co + malware

- okay, now for a redeploy! (CRUD)

```
$ scheduler update t_co t_co.config
Updating job t_co
Restarting shards ...
Getting status ...
Failed Shards = []
...
```

rolling updates …



datacenter operating system

Mesos
+ Twitter specific scheduler
+ service proxy (naming)
+ updater
+ dependency manager

datacenter operating system (private cloud)

Thanks!