DC/OS AND FAST DATA
(THE SMACK STACK)

Benjamin Hindman - @benh
Elizabeth K. Joseph - @pleia2
ARCHITECTURAL SHIFT

TRADITIONAL APPLICATION

Latency

Service

Data

Users

MODERN APPLICATION

3+ Billion internet & smartphone users

Data growth (CAGR): 40%+

Source: KPCB Internet Trends 2016, EMC Digital Universe 2014
TODAY’S REINFORCING TRENDS

CONTAINERIZATION

MICROSERVICES

CONTAINER ORCHESTRATION

BIG DATA & ANALYTICS
TODAY’S REINFORCING TRENDS

- MICROSERVICES
- CONTAINERIZATION
- CONTAINER ORCHESTRATION
- FAST BIG DATA & ANALYTICS
FROM BIG DATA TO FAST DATA

**Days**  **Hours**  **Minutes**  **Seconds**  **Microseconds**

**Batch**  **Micro-Batch**  **Event Processing**

Reports what has happened using descriptive analytics  
Solves problems using predictive and prescriptive analytics

Billing, Chargeback  
Product recommendations  
Real-time Pricing and Routing  
Real-time Advertising  
Predictive User Interface
ON THE EDGE, AND STILL REALLY BIG!

**A380-1000**: 10,000 sensors in each wing; produces more than 7Tb of IoT data per day

[1] https://goo.gl/2S4q5N
MODERN APPLICATION -> FAST DATA BUILT-IN

Use Cases:
- Anomaly detection
- Personalization
- IoT Applications
- Predictive Analytics
- Machine Learning
THE FOUNDATIONS OF FAST DATA

Sensors & Sources

Message Queue

Data Processing

Modern Apps

© 2017 Mesosphere, Inc. All Rights Reserved.
**Message Queues**

**Message Brokers**
- Apache Kafka
- ØMQ, RabbitMQ, Disque

**Log-based Queues**
- fluentd, Logstash, Flume

*see also* queues.io
Typical Use: A reliable buffer for stream processing

Why Kafka?

- High-throughput, distributed, persistent publish-subscribe messaging system
- Created by LinkedIn; used in production by 100+ web-scale companies [1]

[1] https://cwiki.apache.org/confluence/display/KAFKA/Powered+By
DELIVERY GUARANTEES

- **At most once**—Messages may be lost but are never redelivered
- **At least once**—Messages are never lost but may be redelivered (Kafka)
- **Exactly once**—Messages are delivered once and only once (this is what everyone actually wants, but no one can deliver!)

Murphy’s Law of Distributed Systems:

*Anything that can go wrong, will go wrong … partially!*
Microbatching
- Apache Spark (Streaming)

Native Streaming
- Apache Flink
- Apache Storm/Heron
- Apache Apex
- Apache Samza
**APACHE SPARK (STREAMING)**

**Typical Use:** distributed, large-scale data processing; micro-batching

**Why Spark Streaming?**
- Micro-batching creates very low latency, which can be faster
- Well defined role means it fits in well with other pieces of the pipeline
DISTRIBUTED STORAGE

NoSQL
- ArangoDB
- mongoDB
- Apache Cassandra
- Apache HBase

SQL
- MemSQL

Filesystems
- Quobyte
- HDFS

Time-Series Datastores
- InfluxDB
- OpenTSDB
- KairosDB
- Prometheus

see also iot-a.info
**APACHE CASSANDRA**

**Typical Use:** No-dependency, time series database

**Why Cassandra?**
- A top level Apache project born at Facebook and built on Amazon’s Dynamo and Google’s BigTable
- Offers continuous availability, linear scale performance, operational simplicity and easy data distribution
A GOOD STACK ...

Use Cases:
- Anomaly detection
- Personalization
- IoT Applications
- Predictive Analytics
- Machine Learning
how do we operate these distributed systems?
most organizations have many stateless independent (micro)services, the\n\textit{distributed systems} I’m talking about here are ...
how do we scale the operations of distributed systems?
SMACK STACK

Apache Spark: distributed, large-scale data processing

Apache Mesos: cluster resource manager

Akka: toolkit for message driven applications

Apache Cassandra: distributed, highly-available database

Apache Kafka: distributed, highly-available messaging system
distributed systems are *hard* to operate
DATA & ANALYTICS
DAY 2 OPS CHALLENGES

- Bare metal storage (or someone else’s problem)
- Drive down job latency and drive up utilization
- Run multiple versions simultaneously
- Upgrade complicated data systems
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

- fault tolerance
- high availability
- latency
- bandwidth
- CPU/mem resources
- ...

= configuration
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

```
INSTALL.SH
#!/bin/bash

pip install "$1" &
easy_install "$1" &
brew install "$1" &
npm install "$1" &
yum install "$1" &
dnf install "$1" &
docker run "$1" &
figaro install "$1" &
apt-get install "$1" &
sudo apt-get install "$1" &
streamcmd apps_update "$1" validate &
git clone https://github.com/"$1"/"$2" cd "$1"/config/make install &
curl "$1" | bash &
```
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

(1) express

(2) orchestrate

© 2017 Mesosphere, Inc. All Rights Reserved.
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

(1) express
(2) orchestrate
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

first, debug ...
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

first, debug ...
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

second, fix (scale, patch, etc)

...
then, debug again ...
finally, write **code** so it never happens again ...
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1
thesis:
distributed systems should (be able to) operate themselves; deploy, monitor, upgrade ...
why:

(1) operators have *inadequate knowledge* of distributed system needs/semantics to make optimal decisions
why:

(1) operators have *inadequate knowledge* of distributed system needs/semantics to make optimal decisions (even after reading the book)
why:

(2) execution needs/semantics can’t easily or efficiently be expressed to underlying system, and vice versa
(1) express

```
#!/bin/bash

pip install "$1" &
easy_install "$1" &
brew install "$1" &
brew cask install "$1" &
yum install "$1" &
dnf install "$1" &
apt-get install "$1" &
docker run "$1" &
login "$1" &
ps aux | grep "$1" | grep -v grep
```

(2) orchestrate
configuration spectrum:

course-grained

fine-grained
configuration spectrum:

course-grained  fine-grained

easiest to express (how most of us would do it), but worst resource utilization
configuration spectrum:

- coarse-grained
- fine-grained

hardest to express (if even possible), but best resource utilization
why can’t Hadoop decide this for me?
applications “operate” themselves on Linux; when an application needs to “scale up” it asks the operating system to allocate more memory or create another thread ...
application

operating system

syscall interface:
- memory allocate
- clone/fork
- create file
- read, write
- ...

© 2017 Mesosphere, Inc. All Rights Reserved.
once upon a time ... before virtual memory

configuration:
- [0x0, 0x1) → calc.exe
- [0x1, 0x4) → winmine.exe
- [0x4, 0x8) → notepad.exe

physical memory

applications take physical memory address range as an input
once upon a time ... before virtual memory

configuration:
- \([0x0, 0x1) \rightarrow \text{calc.exe}\)
- \([0x1, 0x4) \rightarrow \text{winmine.exe}\)
- \([0x4, 0x8) \rightarrow \text{notepad.exe}\)
how: distributed systems need *interface* to *communicate* with underlying system, *and vice versa*
distributed system

(operating) system

interface:
resource allocation
launch container/VM
create storage
attach/detach storage
...
vice versa: operating system should be able to *callback* into application
learning from history ... bidirectional interface

application

operating system

callback interface:
paging/swapping
CPU deallocation
...

© 2017 Mesosphere, Inc. All Rights Reserved.
learning from history ... bidirectional interface

callback interface: paging/swapping CPU deallocation ...

better than LRU, ask the application what pages to swap!
learning from history … bidirectional interface

callback interface:
- paging/swapping
- CPU deallocation
...

search for ‘scheduler activations’ and ‘Lithe composition’
distributed systems need bidirectional interface too

distributed system

(operating) system

callback interface:
container/VM failed resource deallocation

...
Distributed systems need bidirectional interface too.

Callback interface:
- Container/VM failed
- Resource deallocation
- Tell the distributed system about "planned failures" (i.e., maintenance)
Apache Mesos
Dogfooding: Apache Spark
reality is people are (already) building software that operates distributed systems ...
common pattern: ad hoc control planes

**goal:** provide *distributed system* as software as a service (SaaS) to the rest of your internal organization or to sell to external organizations

**solution:** a *control plane* built out of ad hoc scripts, ancillary services, etc, that deploy, maintain, and upgrade said SaaS

* e.g., analytics via Spark, message queue via Kafka, key/value store via Cassandra
$ kubectl create -f $LOC/kitchensink-master-service.json
$ kubectl create -f $LOC/kitchensink-slave-service.json
$ kubectl create -f $LOC/kitchensink-pgpool-service.json
$ envsubst < $LOC/kitchensink-sync-slave-pv.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-master-pv.json | kubectl create -f -
$ kubectl create -f $LOC/kitchensink-sync-slave-pvc.json
$ kubectl create -f $LOC/kitchensink-master-pvc.json
$ envsubst < $LOC/kitchensink-master-pod.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-slave-dc.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-sync-slave-pod.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-pgpool-rc.json | kubectl create -f -
$ kubectl create -f $LOC/kitchensink-watch-sa.json
$ envsubst < $LOC/kitchensink-watch-pod.json | kubectl create -f -

$ kubectl create -f $LOC/kitchensink-master-service.json
$ kubectl create -f $LOC/kitchensink-slave-service.json
$ kubectl create -f $LOC/kitchensink-pgpool-service.json
$ envsubst < $LOC/kitchensink-sync-slave-pv.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-master-pv.json | kubectl create -f -
$ kubectl create -f $LOC/kitchensink-sync-slave-pvc.json
$ kubectl create -f $LOC/kitchensink-master-pvc.json
$ envsubst < $LOC/kitchensink-master-pod.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-slave-dc.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-sync-slave-pod.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-pgpool-rc.json | kubectl create -f -
$ kubectl create -f $LOC/kitchensink-watch-sa.json
$ envsubst < $LOC/kitchensink-watch-pod.json | kubectl create -f -
what happens if there’s a bug in the control plane?

what if my control plane has diverged from yours?

what happens when a new release of the distributed system invalidates an assumption the control plane previously made?
control planes should be built into the distributed systems itself by the experts who built the distributed system in the first place!

as an industry we should strive to build a standard interface that distributed systems can leverage
vice versa:

abstractions exist for good reasons, but without sufficient communication they force sub-optimal outcomes …
control planes should be built into distributed systems themselves by the experts who built the distributed system in the first place!

as an industry we should strive to build a standard interface distributed systems can leverage

our standard interface should be bidirectional to avoid sub-optimal outcomes
how do we scale the operations of distributed systems?
let them *scale* themselves!
OPERATING SYSTEMS ARE FOR APPLICATIONS

“SaaS” Experience using DC/OS

Spark  DataStax Enterprise  GitLab  Elasticsearch  HDFS
Jenkins  Confluent Kafka  Cassandra  MariaDB  Zeppelin
Riak  ArangoDB  JFrog Artifactory  Storm  MemSQL

DC/OS
DC/OS SERVICE MANAGES IT'S OWN UPGRADES
<table>
<thead>
<tr>
<th>CAPABILITY</th>
<th>AWS</th>
<th>AZURE</th>
<th>GCP</th>
<th>DC/OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>Object Storage</td>
<td>S3</td>
<td>Blob Storage</td>
<td>Cloud Storage</td>
</tr>
<tr>
<td></td>
<td>Block Storage</td>
<td>Elastic Block Storage (EBS)</td>
<td>Page Blobs, Premium Storage</td>
<td>GCE Persistent Disks</td>
</tr>
<tr>
<td></td>
<td>File Storage</td>
<td>Elastic File System</td>
<td>File Storage</td>
<td>ZFS / Avere</td>
</tr>
<tr>
<td>DB</td>
<td>Relational</td>
<td>RDS</td>
<td>SQL Database</td>
<td>Cloud SQL (MySQL)</td>
</tr>
<tr>
<td></td>
<td>NoSQL</td>
<td>DynamoDB</td>
<td>DocumentDB</td>
<td>Datastore, Bigtable</td>
</tr>
<tr>
<td>Data &amp; Analytics</td>
<td>Full Text Search</td>
<td>CloudSearch</td>
<td>Log Analytics, Search</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Hadoop / Analytics</td>
<td>Elastic Map Reduce (EMR)</td>
<td>HDInsight</td>
<td>Dataproc, Dataflow</td>
</tr>
<tr>
<td></td>
<td>Stream Processing / Ingest</td>
<td>Kinesis</td>
<td>Stream Analytics, Data Lake</td>
<td>Kinesis</td>
</tr>
<tr>
<td></td>
<td>Data Warehouse</td>
<td>Redshift</td>
<td>SQL Data Warehouse</td>
<td>BigQuery</td>
</tr>
<tr>
<td>Other</td>
<td>Monitoring</td>
<td>CloudWatch</td>
<td>Application Insights, Portal</td>
<td>Stackdriver Monitoring</td>
</tr>
<tr>
<td></td>
<td>Serverless</td>
<td>Lambda</td>
<td>Azure Functions</td>
<td>Google Cloud Functions</td>
</tr>
</tbody>
</table>
THANK YOU!

DEMO!

QUESTIONS?

@dcos
chat.dcos.io
users@dcos.io
/groups/8295652
/dcoss
/dcoss/examples
/dcoss/demos
bigger picture:

abstractions exist for good reasons, but without sufficient communication they force sub-optimal outcomes ...