Using the SDACK Architecture to Build a Big Data Product

Yu-hsin Yeh (Evans Ye)
Apache Big Data NA 2016
Vancouver
Outline

• A Threat Analytic Big Data product
• The SDACK Architecture
• Akka Streams and data pipelines
• Cassandra data model for time series data
Who am I

- Apache Bigtop PMC member
- Software Engineer @ Trend Micro
- Develop big data apps & infra
A Threat Analytic
Big Data Product
Target Scenario

Security information and event management (SIEM)
Problems

• Too many log to investigate

• Lack of actionable, prioritized recommendations
Threat Analytic System

- Prioritization
- Filtering
- Anomaly Detection
- Expert Rule
- Notification

- splunk
- ArcSight

AD
Web Proxy
Windows event
DNS
...
Deployment
On-Premises

Security information and event

splunk

ArcSight

Prioritization
Filtering
Anomaly Detection
Expert Rule

InfoSec Team

Enterprise
On-Premises

AD

Windows Event Log

DNS

\[\ldots\]\n
On-Premises InfoSec Team

Anomaly Detection

Filtering

Prioritization

Expert Rule

ENTERPRISE

InfoSec Team
On the cloud?

- AD
- Windows Event Log
- DNS

InfoSec Team

ENTERPRISE
There’re too many PII data in the log
System Requirement

- Identify anomaly activities based on data
- Lookup extra info such as ldap, active directory, hostname, etc
- Correlate with other data
  - EX: C&C connection followed by an anomaly login
- Support both **streaming** and **batch** analytic workloads
  - Run streaming detections to identify anomaly in timely manner
  - Run batch detections with arbitrary start time and end time
- Scalable
- Stable
How exactly are we going to build this product?

No problem. I have a solution.
SMACK Stack
Toolbox for wide variety of data processing scenarios
SMACK Stack

- **Spark**: fast and general purpose engine for large-scale data processing scenarios
- **Mesos**: cluster resource management system
- **Akka**: toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications
- **Cassandra**: distributed, highly available database designed to handle large amounts of data across datacenters
- **Kafka**: high-throughput, low-latency distributed pub-sub messaging system for real-time data feeds

Medium-sized Enterprises

- AD
- Windows Event Log
- DNS

Spark, akka, cassandra, kafka

InfoSec Team

ENTERPRISE

TREND MICRO
Can’t we simplify it?

No problem. I have a solution.
SDACK Stack
SDACK Stack

- **Spark**: fast and general purpose engine for large-scale data processing scenarios
- **Docker**: deployment and resource management
- **Akka**: toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications
- **Cassandra**: distributed, highly available database designed to handle large amounts of data across datacenters
- **Kafka**: high-throughput, low-latency distributed pub-sub messaging system for real-time data feeds

Docker - How it works

• Three key techniques in Docker

  • Use Linux Kernel’s **Namespaces** to create isolated resources

  • Use Linux Kernel’s **cgroups** to constrain resource usage

  • **Union file system** that supports git-like image creation
The nice things about it

• Lightweight
• Fast creation
• Efficient to ship
• Easy to deploy
• Portability
  • runs on *any* Linux environment
Medium-sized Enterprises

- AD
- Windows Event Log
- DNS

InfoSec Team

ENTERPRISE

TREND MICRO™
Large Enterprises

- AD
- Windows Event Log
- DNS

→

→

→

InfoSec Team

ENTERPRISE
Threat Analytic System Architecture
Docker Compose

kafka:
  build: .
  ports:
    - “9092:9092”

spark:
  image: spark
  port:
    - “8080:8080”

......
Akka Streams and data pipelines
Why not Spark Streaming?

- Each streaming job occupies at least 1 core
- There’re so many type of log
  - AD, Windows Event, DNS, Proxy, Web Application, etc
- Kafka offset loss when code changed
- No back-pressure (back then)
  - Estimated back-pressure available in Spark 1.5
- Integration with Akka based info lookup services
• High performance concurrency framework for Java and Scala

• Actor model for message-driven processing

• Asynchronous by design to achieve high throughput

• Each message is handled in a single threaded context (no lock, synchronous needed)

• Let-it-crash model for fault tolerance and auto-healing

• Clustering mechanism available
Akka Streams

• Akka Streams is a DSL library for doing streaming computation on Akka

- **Source** → **Flow** → **Sink**

```scala
def Source(1 to 3).
  .map { i => println(s"A: $i"); i }
  .map { i => println(s"B: $i"); i }
  .map { i => println(s"C: $i"); i }
  .runWith(Sink.ignore)
```

• Materializer to transform each step into Actor

```scala
implicit val materializer = ActorMaterializer()
```

• Back-pressure enabled by default
Reactive Kafka

- Akka Streams wrapper for Kafka
- Commit processed offset back into Kafka

```scala
Source(kafkaConsumer.publisher)
  .via(balancedFlow)
  .to(kafkaConsumer.offsetCommitSink).run()
```

- Provide **at-least-once** message delivery guarantee

https://github.com/akka/reactive-kafka
Message Delivery Semantics

- **Actor Model**: at-most-once

- **Akka Persistence**: at-least-once
  - Persist log to external storage (like WAL)

- **Reactive Kafka**: at-least-once + back-pressure
  - Write offset back into Kafka

- **At-least-once + Idempotent writes = exactly-once**
Implementing Data Pipelines

- Kafka source
- Parsing Phase
- Storing Phase
- Manual commit

Reactive Kafka
Akka Stream
Scale Up

- Kafka Source
- Parsing Phase
- Storing Phase
- Manual Commit
- Map Async
- Router
- Parser
- ...
Scale Out

- Scale out by creating more docker containers with same **Kafka consumer group**
Be careful using Reactive Kafka Manual Commit
A bug that periodically reset the committed offset

<table>
<thead>
<tr>
<th>#partition</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition 0</td>
<td>50</td>
</tr>
<tr>
<td>partition 1</td>
<td>90</td>
</tr>
<tr>
<td>partition 2</td>
<td>70</td>
</tr>
</tbody>
</table>
A bug that periodically reset committed offset

<table>
<thead>
<tr>
<th>partition</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition 0</td>
<td>300</td>
</tr>
<tr>
<td>partition 1</td>
<td>250</td>
</tr>
<tr>
<td>partition 2</td>
<td>70</td>
</tr>
<tr>
<td>partition 2</td>
<td>350</td>
</tr>
</tbody>
</table>
Normal: redline(lag) keep declining
Abnormal: redline(lag) keep getting reset
PR merged in 0.8 branch
Apply patch if version <= 0.8.7

https://github.com/akka/reactive-kafka/pull/144/commits
Akka Cluster

• Connects Akka runtimes into a cluster

• No-SPOF

• Gossip Protocol

• ClusterSingleton library available
Lookup Service Integration

Akka Streams

Akka Cluster

ClusterSingleton

Ldap Lookup

Hostname Lookup
Lookup Service Integration

Akka Streams

ClusterSingleton

Ldap Lookup

Hostname Lookup

Akka Cluster
Lookup Service Integration

Hostname Lookup

Ldap Lookup

Service Discovery

ClusterSingleton

Akka Streams

Akka Streams

Akka Streams

Akka Cluster
Async Lookup Process

- Synchronous
- Asynchronous

Diagram:
- Hostname Lookup
- ClusterSingleton
- Cache
- Akka Streams
Async Lookup Process

Synchronous
Asynchronous

Hostname Lookup
ClusterSingleton
Cache
Akka Streams

PostgreSQL
Async Lookup Process

- Synchronous
- Asynchronous

- Hostname Lookup
- ClusterSingleton
- Cache
- Akka Streams

PostgreSQL
Cassandra Data Model for Time Series Data
Event Time V.S. Processing Time
Problem

- We need to do analytics based on event time instead of processing time
  - Example: modelling user behaviour
- Log may arrive late
  - network slow, machine busy, service restart, etc
- Log arrived out-of-order
  - multiple sources, different network routing, etc
Problem

• Spark streaming can’t handle event time
Solution

• Store time series data into Cassandra
• Query Cassandra based on event time
• Get most accurate result upon query executed
• Late tolerance time is flexible
• Simple
Cassandra Data Model

\[\text{Map<RowKey, SortedMap<ColumnKey, ColumnValue>>}\]

- Retrieve an entire row is fast
- Retrieve a range of columns is fast
  - Read continuous blocks on disk (not random seek)
Data Model for AD log

// AD table
CREATE TABLE threat_analytic_engine.ad (  
  shardId int,  
  DDMMYYHH text,  
  rt bigint,  
  UUID uuid,  
  accountName text,  
  clientAddr text,  
  computerName text,  
  domainName text,  
  eventId text,  
  logonId text,  
  peerHost text,  
  resCode text,  
  PRIMARY KEY ((shardId, DDMMYYHH), rt, UUID)  
) WITH CLUSTERING ORDER BY (rt DESC);
Data Model for AD log

// AD table
CREATE TABLE threat_analytic_engine.ad (  
  shardId int,  
  DDMMYYHH text,  
  rt bigint,  
  UUID uuid,  
  accountName text,  
  clientAddr text,  
  computerName text,  
  domainName text,  
  eventId text,  
  logonId text,  
  peerHost text,  
  resCode text,  
  PRIMARY KEY (((shardId, DDMMYYHH), rt, UUID))  
) WITH CLUSTERING ORDER BY (rt DESC);
Data Model for AD log

```
// AD table
CREATE TABLE threat_analytic_engine.ad (  
  shardId int,
  DDMYYYHH text,
  rt bigint,
  UUID uuid,
  accountName text,
  clientAddr text,
  computerName text,
  domainName text,
  eventId text,
  logonId text,
  peerHost text,
  resCode text,
  PRIMARY KEY ((shardId, DDMYYYHH), rt, UUID)  
) WITH CLUSTERING ORDER BY (rt DESC);
```

clustering key for range queries
Wrap up
Recap: SDACK Stack

- **Spark**: streaming and batch analytics
- **Docker**: ship, deploy and resource management
- **Akka**: resilient, scalable, flexible data pipelines
- **Cassandra**: batch queries based on event data
- **Kafka**: durable buffer, streaming processing
Thank you!

Questions?
Backup
Query APIs

• val conn = new CassandraDataConnector(sc)

  val rdd = conn
  .getData(timeFrame, table))

val rdd = conn
  .getData(timeFrame, table, eventid, clientaddr))

• val rdd = sc.cassandraTable(keyspace, table)
  .select(columns: _*)
  .where("shardid in (?, ?, ?)", 0, 1, 2)
  .where("ddmmyyhh in (?)", "09-05-2016-00")
Sync Lookup

- Synchronous
- Asynchronous

Hostname Lookup

ClusterSingleton

Cache

Akka Streams

PostgreSQL
No Back-pressure

Source → Fast!!! → Slow… → Sink

\[ v(\_\_\_)y \quad (>\_\_\_<)’ \]
No Back-pressure

Source → Fast!!! → Slow… → Sink

v(owo)y (>/.<)’’’’
With Back-pressure
With Back-pressure

Source \(\rightarrow\) Fast!!! \(\rightarrow\) Slow… \(\rightarrow\) Sink

request 3 \(\rightarrow\) request 3
• Distributed NoSQL key-value storage, no SPOF

• Fast on write, suitable for time series data

• Decent read performance, if designed right
  • Build data model around your queries

• Spark Cassandra Connector

• Configurable CA (CAP theorem)
  • Choose A over C and vise-versa

Dynamo: Amazon’s Highly Available Key-value Store
fluentd -> receiver
fluentd -> kafka
kafka -> buffer
kafka -> akka
akka -> transformation, lookup ext info
akka -> kafka
akka -> cassandra
kafka -> Spark
Spark -> batch
Spark -> streaming
Sweet Spots

- Choose availability over consistency
  - We don’t do update, hence no strong consistency required
- Native, efficient and flexible time range queries
- CQL to support research and log investigations