Software Architecture Extraction: for Stream Processing and Batch Systems

Pooyan Jamshidi
Imperial College London
p.jamshidi@imperial.ac.uk

SMA: Software Modeling and Analysis
University of Bern, Switzerland
2nd Nov 2016
Agenda

• Preliminary concepts about Software Architecture
• Temporal aspect of Software Architecture
• Data-Intensive Computing
  • Stream Processing: Apache Storm
  • Batch Processing: Apache Hadoop
• OSTIA: a solution for architectural extraction of Big Data
• Lab
We need some preliminary concept on software architecture (SA) to better understand architecture recovery concepts.
Software Architecture’s Elements

- A software system’s architecture typically is not (and should not be) a uniform monolith
- A software system’s architecture should be a composition and interplay of different elements
  - Processing
  - Data, also referred as information or state
  - Interaction

A software architecture model consists of different components and their interactions and may be represented through different views.
Although SA can be represented in different view and each view of SA serve different stakeholders, here we concentrate on logical view (C&C) which can be useful for developers providing useful information for developers in the software development process.
Components

- Elements that encapsulate processing and data in a system’s architecture are referred to as *software components*.
- Definition: A *software component* is an architectural entity that:
  - encapsulates a subset of the system’s functionality and/or data
  - restricts access to that subset via an explicitly defined interface
  - has explicitly defined dependencies on its required execution context
- Components typically provide application-specific services.
Connectors

• In complex systems, _interaction_ may become more important than the functionality of the components

• Definition
  ➢ A _software connector_ is an architectural building block tasked with effecting and regulating interactions among components

• In many software systems connectors are usually simple procedure calls or shared data accesses
  ➢ Much more sophisticated and complex connectors are possible!
  ➢ There exist languages (Reo) for specifying connectors, and there exists some work about connector evolution and adaptation

• Connectors typically provide application-independent interaction facilities
Examples of Connectors

- Procedure call connectors
- Shared memory connectors
- Message passing connectors
- Streaming connectors
- Distribution connectors
- Wrapper/adaptor connectors
- Coordinator
Interconnection semantics (connector behavior) can be formally specified using a modeling language (state chart, Reo, etc)
Configurations

• Components and connectors are composed in a specific way in a given system’s architecture to accomplish that system’s objective

• Definition
  
  ➢ *An architectural configuration*, or topology, is a set of specific associations between the components and connectors of a software system’s architecture
Every software system undergo different changes. Therefore, it is important to understand how SA can contribute to the software evolution process.
Modern software systems are increasingly required to operate in an open world, characterized by frequent and unpredictable change in the environment in which they are functioning and in the requirements they have to meet. Considering existing research and practice, software architectures provide a sound basis to smoothly evolve software and dynamically adapt it to provide expected services. Architecture-centric software evolution allows an appropriate abstraction to model, analyze and execute software evolution in a controllable and manageable fashion. For more information, please see "A Framework for Classifying and Comparing Architecture-Centric Software Evolution Research".

Temporal Aspect of Software Architecture

• Design decisions are made over a system’s lifetime → Architecture has a temporal aspect
• At any given point in time, the system has only one architecture
• A system’s architecture will change over time
Prescriptive vs. Descriptive Architecture

A system’s *prescriptive architecture* captures the design decisions made prior to the system’s construction
- It is the *as-conceived* or *as-intended* architecture

A system’s *descriptive architecture* describes how the system has been built
- It is the *as-implemented* or *as-realized* architecture

Which architecture is “correct”?
Are the two architectures consistent with one another?
What criteria are used to establish the consistency between the two architectures?
On what information is the answer to the preceding questions based?
A large number of prescriptive and descriptive architectures are created during the lifespan of a typical software system. When a system is initially built or the already implemented system is evolved, its prescriptive architecture is modified appropriately followed by corresponding changes to its descriptive architecture. But, in practice, the system is often directly modified without taking into account the impact to the perspective architecture when it is not modified. The failure to update the prescriptive architecture results in potential dangers especially if the software systems are bound to contain many errors. The resulting discrepancy between a system’s prescriptive and descriptive architecture is referred to as architectural degradation.

Architectural degradation comprises of two related phenomena -
Architectural drift: is introduction of principal design decisions into a system’s descriptive architecture that are not included in, encompassed by or implied by the prescriptive architecture, but which do not violate any of the prescriptive architecture design decisions.
Architectural erosion: is the introduction of architectural design decisions into a system’s descriptive architecture that violates its prescriptive architecture.
Reality in practice

• Missing architectural specifications

We need
• Re-documentation
• Reengineering

Frequently we are asked to analyze a system’s software architecture and are given only its code and the (limited) time of a designer. [Kazman et al.’99]
A large number of prescriptive and descriptive architectures are created during the lifespan of a typical software system. When a system is initially built or the already implemented system is evolved, its prescriptive architecture is modified appropriately followed by corresponding changes to its descriptive architecture. But, in practice, the system is often directly modified without taking into account the impact to the perspective architecture when it is not modified. The failure to update the prescriptive architecture results in potential dangers especially if the software systems are bound to contain many errors. The resulting discrepancy between a system’s prescriptive and descriptive architecture is referred to as architectural degradation.

Architectural degradation comprises of two related phenomena -
Architectural drift: is introduction of principal design decisions into a system’s descriptive architecture that are not included in, encompassed by or implied by the prescriptive architecture, but which do not violate any of the prescriptive architecture design decisions.
Architectural erosion: is the introduction of architectural design decisions into a system’s descriptive architecture that violate its prescriptive architecture.

A bit less formal descriptions of degradation:

Architectural erosion usually occurs when a program’s code is initially less than optimal, contains “hacks” to quickly add functionality or simply cannot support the desired new changes due to the code structure. The problems it presents is an ever increasing maintenance cost due to the complexity of the system as a result of the accumulation of various design decisions. The only viable solution to fix or prevent architectural erosion is to rewrite the entire code base from scratch and try to anticipate future developments in order to accommodate them.

Architectural drift is when the implementation of a program diverges from
the initial design and purpose. The problems it brings are similar to erosion; it will be increasingly difficult to further develop, maintain or even understand the code because design decisions are not always apparent when only looking at the code. To prevent architectural drift, there needs to be some kind of communication between the system designers and the implementers such as having the designers follow the progress of the implementation or having the implementers carefully document their work.

As for which is "worse", I think erosion would cost more money to fix because drift can be corrected earlier on in the development stage whereas erosion is usually left until much later, a situation which can be likened to prevention of a problem versus curing the problem after the fact. Also, because erosion is usually dealt with after the development of a program is complete it may involve many more stakeholders than drift would.
What is architecture extraction (recovery)?

- Extracting architecturally significant elements from lower level artifacts.
- Why we need architecture recovery?
  - The design documents are not available
  - If they are available, they may not be in synchronization with the current version of the system

Software architecture recovery is a set of methods for the extraction of architectural information from lower level representations of a software system, such as source code. The abstraction process to generate architectural elements frequently involves clustering source code entities (such as files, classes, functions etc.) into subsystems according to a set of criteria that can be application dependent or not. Architecture recovery from legacy systems is motivated by the fact that these systems do not often have an architectural documentation, and when they do, this documentation is many times out of synchronization with the implemented system.
Architectural Recovery

• If architectural degradation is allowed to occur, one will be forced to recover the system’s architecture sooner or later
• Architectural recovery is the process of determining a software system’s architecture from its implementation-level artifacts
• Implementation-level artifacts can be
  ➢ Source code
  ➢ Executable files
  ➢ Java .class files
There is a need for enforcing the architecture of a system. Here is an illustration of the descriptive and prescriptive architecture and the relations with environments and code. The ideal case is that architecture model is used to enforce any change to the system and it is kept in sync with implementation artifacts at any time. Also the history of changes in the architecture will be versioned and store with the key change decisions, etc.
Challenges

• What process can support uncovering the software architecture within a system?
• How much can you automate in this process?
• What are the limits of architecture recovery? (e.g., recovering *all design decisions*).
For illustration purposes and only due to my experience with big data system, here I concentrate on architecture recovery of big data application in DevOps context. Therefore, before presenting our solution for architecture recovery of big data, I will review the technologies that are supported by our tool. Architecture recovery solutions are typically technology dependent. However, they are traditionally introduced for enterprise applications, but due to the increase popularity of big data applications, we only concentrated on these technologies, however, our solution is extensible for other technologies as well.
Stream Processing: Apache Storm

[Content mainly taken from tutorial on Storm by Michael G. Noll]
Storm?

“Distributed and fault-tolerant real-time computation”
http://storm.incubator.apache.org/
Originated at BackType/Twitter, open sourced in late 2011
Implemented in Clojure, some Java
12 core committers, plus ~ 70 contributors

https://github.com/apache/incubator-storm/#committers
https://github.com/apache/incubator-storm/graphs/contributors
A topology in Storm wires data and functions via a DAG.

Executes on many machines like a MR job in Hadoop.

DAG = Directed Acyclic Graph
The logic for a real-time application is packaged into a Storm topology. A Storm topology is analogous to a MapReduce job. One key difference is that a MapReduce job eventually finishes, whereas a topology runs forever (or until you kill it, of course). A topology is a graph of spouts and bolts that are connected with stream groupings.
Topology

data
Topology

functions

data
A Storm application is designed as a "topology" in the shape of a directed acyclic graph (DAG) with spouts and bolts acting as the graph vertices. Edges on the graph are named streams and direct data from one node to another. Together, the topology acts as a data transformation pipeline. At a superficial level the general topology structure is similar to a MapReduce job, with the main difference being that data is processed in real time as opposed to in individual batches. Additionally, Storm topologies run indefinitely until killed, while a MapReduce job DAG must eventually end. [https://en.wikipedia.org/wiki/Storm_(event_processor)]
Relation of topologies to FP

data

\[ f \]

\[ h \]

Spout 1 \rightarrow Bolt 1 \rightarrow Bolt 4

Spout 2 \rightarrow Bolt 2 \rightarrow Bolt 3
Relation of topologies to FP

DAG: $h(f(data), g(data))$
Previous WordCount example in Storm (high-level)

Remember?
(\rightarrow queries (map second) frequencies (sort-by val >))

queries \quad f \quad g \quad h

\begin{center}
\begin{tikzpicture}
\node[draw] (spout) {Spout};
\node[draw] (bolt1) [right of=spout] {Bolt 1};
\node[draw] (bolt2) [right of=bolt1] {Bolt 2};
\node[draw] (bolt3) [right of=bolt2] {Bolt 3};
\draw[->] (spout) -- (bolt1);
\draw[->] (bolt1) -- (bolt2);
\draw[->] (bolt2) -- (bolt3);
\end{tikzpicture}
\end{center}
Data model

**Tuple** = datum containing 1+ fields

(1.1.1.1, “foo.com”)

Values can be of any type such as Java primitive types, String, byte[]. Custom objects should provide their own Kryo serializer though.

**Stream** = unbounded sequence of tuples

...  
(1.1.1.1, “foo.com”)  
(2.2.2.2, “bar.net”)  
(3.3.3.3, “foo.com”)  
...

Spouts and bolts

Spout = source of data streams

Can be “unreliable” (fire-and-forget) or “reliable” (can replay failed tuples).
Example: Connect to the Twitter API and emit a stream of decoded URLs.

Bolt = consumes 1+ streams and potentially produces new streams

Can do anything from running functions, filter tuples, joins, talk to DB, etc.
Complex stream transformations often require multiple steps and thus multiple bolts.

Stream groupings control the data flow in the DAG

- **Shuffle grouping** = random; typically used to distribute load evenly to downstream bolts
- **Fields grouping** = GROUP BY field(s)
- **All grouping** = replicates stream across all the bolt’s tasks; use with care
- **Global grouping** = stream goes to a single one of the bolt’s tasks; don’t overwhelm the target bolt!
- **Direct grouping** = producer of the tuple decides which task of the consumer will receive the tuple
- **LocalOrShuffle** = if the target bolt has one or more tasks in the same worker process, tuples will be shuffled to just those in-process tasks. Otherwise, same as normal shuffle.

- Custom groupings are possible, too.
Worker processes vs. Executors vs. Tasks

A machine in a Storm cluster may run one or more worker processes for one or more topologies. Each worker process runs executors for a specific topology.

One or more executors may run within a single worker process, with each executor being a thread spawned by the worker process. Each executor runs one or more tasks of the same component (spout or bolt).

A task performs the actual data processing.

A worker process is either idle or being used by a single topology, and it is never shared across topologies. The same applies to its child executors and tasks.

Storm distinguishes between the following three main entities that are used to actually run a topology in a Storm cluster:

- **Worker processes**
- **Executors (threads)**
- **Tasks**

A **worker process** executes a subset of a topology, and runs in its own JVM. A worker process belongs to a specific topology and may run one or more executors for one or more components (spouts or bolts) of this topology. A running topology consists of many such processes running on many machines within a Storm cluster.

An **executor** is a thread that is spawned by a worker process and runs within the worker’s JVM. An executor may run one or more tasks for the same component (spout or bolt). An executor always has one thread that it uses for all of its tasks, which means that tasks run serially on an executor.

A **task** performs the actual data processing and is run within its parent executor’s thread of execution. Each spout or bolt that you implement in your code executes as many tasks across the cluster. The number of tasks for a component is always the same throughout the lifetime of a topology, but the number of executors (threads) for a component can change over time. This means that the following condition holds true: #threads <= #tasks. By default, the number of tasks is set to be the same as the number of executors, i.e. Storm will run one task per thread (which is usually what you want anyways).

The following illustration shows how a simple topology would look like in operation. The topology consists of three components: one spout called BlueSpout and two bolts called GreenBolt and YellowBolt. The components are linked such that BlueSpout sends its output to GreenBolt, which in turns sends its own output to YellowBolt.

The GreenBolt was configured as per the code snippet above whereas BlueSpout and YellowBolt only set the parallelism hint (number of executors). See the code in the next slide...
Code to configure this topology

```java
Config conf = new Config();
conf.setNumWorkers(2); // use two worker processes

TopologyBuilder.setSpout("blue-spout", new Bluespout(), 2); // parallelism hint

TopologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
  .setNumTasks(1)
  .shuffleGrouping("blue-spout");

TopologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 3)
  .shuffleGrouping("green-bolt");

StromSubmitter.submitTopology("mytopology", conf, topologyBuilder.createTopology());
```
# Storm architecture

<table>
<thead>
<tr>
<th>Hadoop v1</th>
<th>Storm</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>JobTracker</td>
<td>Nimbus (only 1)</td>
<td>- distributes code around cluster&lt;br&gt;- assigns tasks to machines/supervisors&lt;br&gt;- failure monitoring&lt;br&gt;- is fail-fast and stateless (you can “kill -9” it)</td>
</tr>
<tr>
<td></td>
<td>Supervisor (many)</td>
<td>- listens for work assigned to its machine&lt;br&gt;- starts and stops worker processes as necessary based on Nimbus&lt;br&gt;- is fail-fast and stateless (you can “kill -9” it)&lt;br&gt;- shuts down worker processes with “kill -9”, too</td>
</tr>
<tr>
<td>MR job</td>
<td>Topology</td>
<td>- processes messages forever (or until you kill it)&lt;br&gt;- a running topology consists of many worker processes spread across many machines</td>
</tr>
</tbody>
</table>
Storm Architecture

ZooKeeper cluster

provides service discovery and coordination

Storm cluster

nimbus1

slave1

slave2

... 

slaveN

nimbus

ui

super
visor

super
visor

super
visor

super
visor
A Storm cluster is superficially similar to a Hadoop cluster. Whereas on Hadoop you run "MapReduce jobs", on Storm you run "topologies". "Jobs" and "topologies" themselves are very different -- one key difference is that a MapReduce job eventually finishes, whereas a topology processes messages forever (or until you kill it).

There are two kinds of nodes on a Storm cluster: the master node and the worker nodes. The master node runs a daemon called "Nimbus" that is similar to Hadoop's "JobTracker". Nimbus is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures.

Each worker node runs a daemon called the "Supervisor". The supervisor listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it. Each worker process executes a subset of a topology; a running topology consists of many worker processes spread across many machines.
A trivial “Hello, Storm” topology

Spout → Belt

“emit random number < 100” → “multiply by 2”

(74) → (148)
**Code**

**Spout**
```java
public void nextTuple()
{
    final Random rand = new Random(); // normally this should be an instance field
    int nextRandomNumber = rand.nextInt(100);
    collector.emit(new Values(nextRandomNumber)); // auto-boxing
}
```

**Bolt**
```java
@override
public void prepare(Map conf, TopologyContext context, OutputCollector collector)
{
    this.collector = collector;
}

@override
public void execute(Tuple tuple)
{
    Integer inputNumber = tuple.getInteger(0);
    collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
    collector.ack(tuple);
}

@override
public void declareOutputFields(OutputFieldsDeclarer declarer)
{
    declarer.declare(new Field("doubled-number"));
}
```
Code

Topology config – for running on your local laptop

```java
Config conf = new Config();
conf.setNumWorkers(1);
topologyBuilder.setSpout("my-spout", new MySpout(), 2);
topologyBuilder.setBolt("my-bolt", new MyBolt(), 2)
    .shuffleGrouping("my-spout");
StormSubmitter.submitTopology("mytopology", conf, topologyBuilder.createTopology());
```
Code

Topology config – for running on a production Storm cluster

```java
Config conf = new Config();
conf.setNumWorkers(100);  
topologyBuilder.setSpout("my-spout", new MySpout(), 100);  
topologyBuilder.setBolt("my-bolt", new MyBolt(), 100).shuffleGrouping("my-spout");  
StormSubmitter.submitTopology("mytopology", conf, topologyBuilder.createTopology());
```
Creating a bolt

Storm is polyglot – but in this workshop we focus on JVM languages.
Two main options for JVM users:
Implement the `IRichBolt` or `IBasicBolt` interfaces
Extend the `BaseRichBolt` or `BaseBasicBolt` abstract classes

`BaseRichBolt`
You must – and are able to – manually `ack()` an incoming tuple.
Can be used to delay acking a tuple, e.g. for algorithms that need to work across multiple incoming tuples.

`BaseBasicBolt`
Auto-acks the incoming tuple at the end of its `execute()` method.
These bolts are typically simple functions or filters.
Extending BaseRichBolt

Let's re-use our previous example bolt.

```java
@Override
public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
  this.collector = collector;
}

@Override
public void execute(Tuple tuple) {
  Integer inputNumber = tuple.getInteger(0);
  collector.emit(tuple, new Values(inputNumber * 2)); // auto-boxing
  collector.ack(tuple);
}

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields("doubled-number"));
}
```
Extending `BaseRichBolt`

`execute()` is the heart of the bolt.

This is where you will focus most of your attention when implementing your bolt or when trying to understand somebody else’s bolt.

```java
@Override
public void prepare(Map conf, TopologyContext context, OutputCollector collector)
{
    this.collector = collector;
}

@Override
public void execute(Tuple tuple) {
    integer inputNumber = tuple.getInteger(0);
    collector.emit(tuple, new Values(inputNumber * 2)); // auto-boring
    collector.ack(tuple);
}

@Override
public void declareOutputFields(OutputFieldsDeclarer declarer)
{
    declarer.declare(new Fields(“doubled-number”));
}
```
Extending BaseRichBolt

`prepare()` acts as a “second constructor” for the bolt’s class. Because of Storm’s distributed execution model and serialization, `prepare()` is often needed to fully initialize the bolt on the target JVM.

```java
@Override
public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
  this.collector = collector;
}

@Override
public void execute(Tuple tuple) {
  Integer inputNumber = tuple.getInteger(0);
  collector.emit(tuple, new Values(inputNumber * 2)); // auto-boring
  collector.ack(tuple);
}

@Override
public void declareOutputFields(OutputFieldsDeclarer declerer) {
  declerer.declare(new Field("doubled-number"));
}
```
Extending BaseRichBolt

`declareOutputFields()` tells downstream bolts about this bolt’s output. What you declare must match what you actually emit().
You will use this information in downstream bolts to “extract” the data from the emitted tuples.
If your bolt only performs side effects (e.g. talk to a DB) but does not emit an actual tuple, override this method with an empty {} method.

```java
1  @Override
2  public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
3      this.collector = collector;
4  }
5  
6  @Override
7  public void execute(Tuple tuple) {
8      int inputNumber = tuple.getInteger(0);
9      collector.emit(tuple, new Values(inputNumber * 2)); // auto-boring
10     collector.ack(tuple);
11  }
12  
13  @Override
14  public void declareOutputFields(OutputFieldsDeclarer declarer) {
15      declarer.declare(new Fields("doubled-number"));
16  }
```
Creating a topology

When creating a topology you’re essentially defining the DAG – that is, which spouts and bolts to use, and how they interconnect.

TopologyBuilder#setSpout() and TopologyBuilder#setBolt()
Groupings between spouts and bolts, e.g. shuffleGrouping()
Creating a topology

You must specify the initial *parallelism* of the topology.
Crucial for P&S but no rule of thumb. We talk about tuning later.
You must understand concepts such as workers/executors/tasks.
Only some aspects of parallelism can be changed later, i.e. at run-time.
You can change the #executors (threads).
You cannot change #tasks, which remains static during the topology’s lifetime.

```java
1. Config conf = new Config();
2. conf.setNumWorkers(100);
3. TopologyBuilder builder = new TopologyBuilder();
4. builder.setSpout("my-spout", new MySpout(), 1000);
5. builder.setBolt("my-bolt", new MyBolt(), 200)
   .shuffleGrouping("my-spout"));
6. StormTopology topology = builder.createTopology();
```
Storm adoption and use cases

Twitter: personalization, search, revenue optimization, ...
200 nodes, 30 topos, 50E msg/day, avg latency <50ms, Jun 2013
Yahoo: user events, content feeds, and application logs
320 nodes (YARN), 130k msg/s, June 2013
Spotify: recommendation, ads, monitoring, ...
v0.8.0, 22 nodes, 15+ topos, 200k msg/s, Mar 2014
Alibaba, Cisco, Flickr, PARC, WeatherChannel, ...
Netflix is looking at Storm and Samza, too.

https://github.com/nathanmarz/storm/wiki/Powered-By
Storm at Spotify

[Image: A cup of coffee with a phone in the background.

Getting Data

- accesspoint
- playlist
- search
- storage
- kafka

Social
Topology

kafka spout

EndSong filter

metadata

metadata decorator

privacy filter

listening trigger

pref

ZMTP publisher

SUB

GET

GET
Batch Processing: Apache Hadoop
HDFS - Hadoop Distributed FS

Hadoop uses HDFS, a distributed file system based on GFS (Google File System), as its shared file system.

Files are divided into large blocks and distributed across the cluster (64MB).
Blocks replicated to handle hardware failure.
Above the file systems comes the MapReduce engine, which consists of one Job Tracker, to which client applications submit MapReduce jobs. The Job Tracker pushes work out to available Task Tracker nodes in the cluster, striving to keep the work as close to the data as possible. With a rack-aware filesystem, the Job Tracker knows which node contains the data, and which other machines are nearby. If the work cannot be hosted on the actual node where the data resides, priority is given to nodes in the same rack. This reduces network traffic on the main backbone network. If a Task Tracker fails or times out, that part of the job is rescheduled. If the Job Tracker fails, all ongoing work is lost.

**Hadoop Architecture**

- **Master-Slave Architecture**
- **HDFS Master “Namenode”**
  - Accepts MR jobs submitted by users
  - Assigns Map and Reduce tasks to Tasktrackers
  - Monitors task and tasktracker status, re-executes tasks upon failure
- **HDFS Slaves “Datanodes”**
  - Run Map and Reduce tasks upon instruction from the Jobtracker
  - Manage storage and transmission of intermediate output
HDFS Architecture

Hadoop HDFS + MR cluster

Client

Submit Job

Get Block Locations

JobTracker

Namenode

HTTP Monitoring UI

Machines with Datanodes and Tasktrackers

T  D  T  D  T  D  T  D  T  D
HDFS has a master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system’s clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.
Running jobs on Hadoop

Wordcount on a huge file

Common pattern for things like Log Processing, Statistics, Index creation, Search Engines!
MapReduce: Data Flow

- User jobs are broken into Map tasks and Reduce tasks
- Data is sequence of keys and values
- Map Task: invokes Mapper
  - Input: key1, value1 pair
  - Output: key2, value2 pairs
- Reduce Task: invokes Reducer
  - Called once per a key, in sorted order
  - Input: key2, stream of value2
  - Output: key3, value3 pairs
Data: Stream of keys and values

Map
<0> Hi how are you
<100> I am good

<0> Hello Hello how are you
<105> Not so good

Reduce
are 1
Hi 1
how 1
You 1

are 1
Hello 1
how 1
You 1

are [1 1]
Hello [1 1]
Hi [1]
how [1 1]
you [1 1]

are 2
Hello 2
Hi 1
how 2
you 2

Input
Intermediate results
Output
OSTIA - On-the-fly Static Topology Inference Analysis
“DevOps is the practice of operations and development engineers participating together in the entire service lifecycle, from design through the development process to production support.”

DevOps is also characterized by operations staff making use many of the same techniques as developers for their systems work.

Bridges the gap between development and operations
Creates a collaborative mindset where a single team performs Dev and Ops
Continuous Architecting and Big-Data

Beyond the tremendous hype and diffusion of Big-Data applications in recent years
• High infrastructure costs
• Steep learning curve for different frameworks
• Complex governance of such complex large scale architectures
Continuous Architecting and Big-Data

(provided a Big Data application)
Supporting continuous and incremental improvement of architectural design by means of
• monitoring on platform and infrastructure.
• a constant stream of analyses on the running applications

Desired benefits:
• Reducing (re-)design efforts
• Accelerating and facilitating (re-)deployability

Need to narrow down the scope...
Research Solution

- Identify common anti-patterns
- Identify possible algorithmic manipulations
- Elicitate structural properties and consistency checks
- Investigate further analysis based on formal verification techniques
- Design a tool to support incremental and iterative refinement of streaming topologies by leveraging the above-mentioned findings
OSTIA - On-the-fly Static Topology Inference Analysis

Goals

- Inferring application architecture through on-the-fly reverse engineering and architecture recovery
- Enact continuous architecting by applying iterative refinement
Anti-pattern Detection

Cycle-in

Persistent Data

Multi-Anchoring

Computational Funnel
Algorithmic Manipulation

Topology Cascading

Linearization
Formal Verification

- Automatic encoding of temporal logic formulae based on topology structure.
- It transparently extends core OSTIA
- Complements the empirical analysis
- Non functional properties based on real-time constraints
Example (1) - storm-focused-crawler

- The storm-focused-crawler deals with the management of the URLs that were extracted from source data collected by the stream-manager.
- It uses Storm and MongoDB to mine data from online sources and elicit essential topics.
- Benefits from visualization
- Relevant refactoring hints from algorithmic analysis endorsed by formal verification
- Configuration optimization on hotspot components

Source code: https://github.com/socialsensor/storm-focused-crawler

Our industrial partner experienced benefits from visualization
- understanding complexity
- visualizing critical components
Architecture extraction helps to formalize, enrich, document the software architecture.

Formal analysis to focus on bottleneck components and use configuration optimization for optimizing the topology throughput and latency.
Example (2) - StormCV

- StormCV enables the use of Apache Storm for video processing by adding computer vision (CV) specific operations and data model.
- The platform enables the development of distributed video processing pipelines which can be deployed on Storm clusters.
Example (2) - StormCV

- Visualization help in reverse-engineering new topologies
- Better understanding of the complexities
- Anti-patterns detection
- Provide a more precise and detailed view of the architecture

Source code: https://github.com/sensorstorm/StormCV
Example (3) - Storm-crawler

The aims of storm-crawler is to help build web crawlers that are:

• scalable
• low latency
• easy to extend
• polite yet efficient

Source code: https://github.com/DigitalPebble/storm-crawler
Architecture extraction can help to provide a better and more accurate view of the underlying software.
The formal analysis of the “focused-crawler” topology confirmed the critical role of the “expander” bolt, previously noticed with the aim of OSTIA visual output. It emerged from the output traces that there exists an execution of the system, even without failures, where the queue occupation level is unbounded. The figure shows how the tool constructed a periodic model in which a suffix (highlighted in red) of a finite sequence of events is repeated infinitely many times after a prefix (in white). After ensuring that the trace is not a spurious model, we concluded that the expander queue, having an increasing trend in the suffix, is unbounded. These types of heavyweight and powerful analyses are made easier by OSTIA in that our tool provides a ready-made analysable counterpart of the elicited topologies making almost invisible the formal verification layer (other than manually setting and tuning operational parameters for verification) on top of which OSTIA support is harnessed.
Conclusions

Approach for supporting reverse engineering and recovering of deployed applications for incremental improvement

- Architecture description
- Software evolution
- Stakeholder discussions (for change decisions)
- Anti-pattern detection
- Algorithmic analysis
- Formal Verification

Qualitative evaluation on open source and in use systems
OSTIA - Storm

1- get the tool: https://github.com/maelstromdat/OSTIA

$ cd ostia-rb
$ ./ostia.rb --topology ../examples/FocusedCrawler.java --output focused_crawler.dot --format dot

$ cd ostia-rb
$ ./ostia.rb --topology ../examples/FocusedCrawler.java --output focused_crawler.json --format json

2- visualize the output dot file, with GraphViz, OmniGraffle, ...

More details about the tool can be found in the wiki and the papers
More details about the tool can be found in the wiki and the papers
OSTIA - Hadoop

More details about the tool can be found in the wiki and the papers
OSTIA - Hadoop

```ruby
$ ostia-hadoop-rb git:(master) ./ostia.rb --topology FocusedCrawler.java --output Focused.dot --format dot

Righe notevoli rilevate

Spouts: 1
Bolts: 8

Riconosciuta struttura storm

setSpout("wpSpout", wpSpout,1)
aggiunto spout:wpSpout
aggiunto bolt:WpDeserializer
aggiunto bolt:expander
aggiunto bolt:articleExtraction
aggiunto bolt:mediaExtraction
aggiunto bolt:webPageUpdater
aggiunto bolt:textIndexer
aggiunto bolt:mediaUpdater
aggiunto bolt:mediatextIndexer

Done
```
Further information


Acknowledgement

Andrea Nocari
Marcello Maria Bersani
Dumian A. Tamburri
Francesco Marconi