



University of Bologna
**Dipartimento di Informatica –
Scienza e Ingegneria (DISI)**
Engineering Bologna Campus

Class of **Computer Networks M** or
**Infrastructures for Cloud
Computing and Big Data**

Global Stream Processing

Luca Foschini

Academic year 2016/2017

Stream Processing

There is more and more interest on stream processing ... so

More and more set of tools are available to express and design a **complex streaming architecture** to be immediately deployed

- **Apache Storm**
- **Yahoo S4**

...

Stream Processing Challenge

- **Large amounts of data** →
Need for **real-time views of data**
 - Social network trends, e.g., Twitter real-time search
 - Website statistics, e.g., Google Analytics
 - Intrusion detection systems, e.g., in most datacenters
- **Process large amounts of data**
with latencies of few seconds
with high throughput

Not MapReduce

The out-of-line workflow is not suitable at all

The typical **Batch Processing** → Need to wait for entire computation on large dataset before completing

In general those approaches are not intended for long-running stream-processing

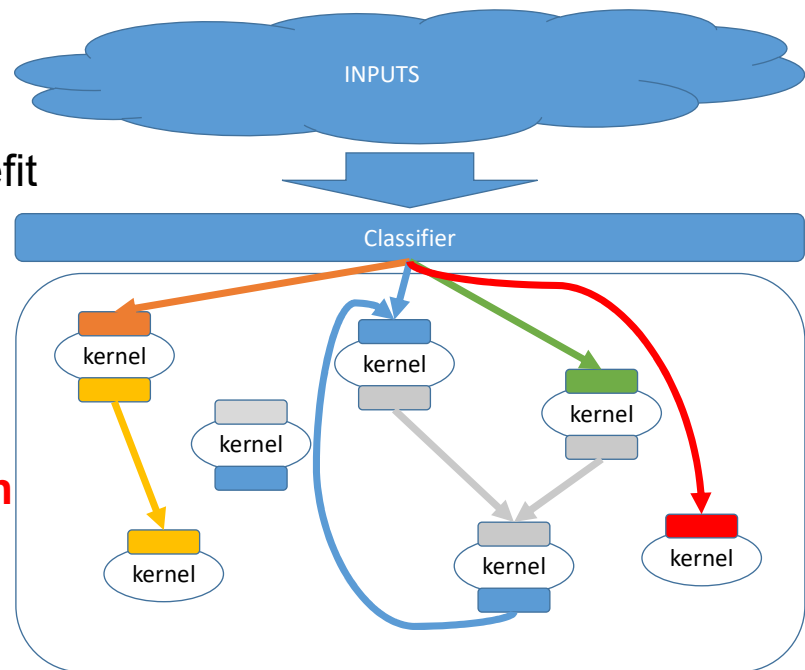
Stream processing model

Stream processing manages:

- Allocation
- Synchronization
- Communication

Applications that benefit most of the streaming model with requirements:

- **High computation resource intensive**
- **Data parallelization**
- **Data time locality**



Stream processing support functions

We need **available some basic functions** that can help in **mapping the concepts** we need to express

Storm is fast in **processing over a million tuples per second per node**: it is **scalable, fault-tolerant, respecting SLA** over data to be processed

Main functions must support the **stream processing** model:

- **Resource allocation**
- **Data classification**
- **Information routing in flows**
- **Management of execution/processing status**

Storm

- **Apache Project** <http://storm.apache.org/>
- Highly active **Java based** JVM project
- **Multiple languages** supported via user API
 - Python, Ruby, etc.
- Over 50 companies use it, including
 - **Twitter**: for personalization, search
 - **Flipboard**: for generating custom feeds
 - Spotify, Groupon, Weather Channel, WebMD, etc.

Storm Core Components

The Storm architecture is based on

- **Tuples**
- **Streams**
- Spouts
- Bolts
- Topologies
- ...

Tuple

The tuple is an ordered list of elements

Tuple

- E.g., <twitter, tweet>
 - E.g., <“Miley Cyrus”, “Hey! Here’s my new song!”>
 - E.g., <“Justin Bieber”, “Hey! Here’s MY new song!”>
- E.g., <URL, clicker-IP, date, time>
 - E.g., <coursera.org, 101.102.103.104, 4/4/2014, 10:35:40>
 - E.g., <coursera.org, 101.102.103.105, 4/4/2014, 10:35:42>

Stream

Tuple

Tuple

Tuple

Sequence of tuples

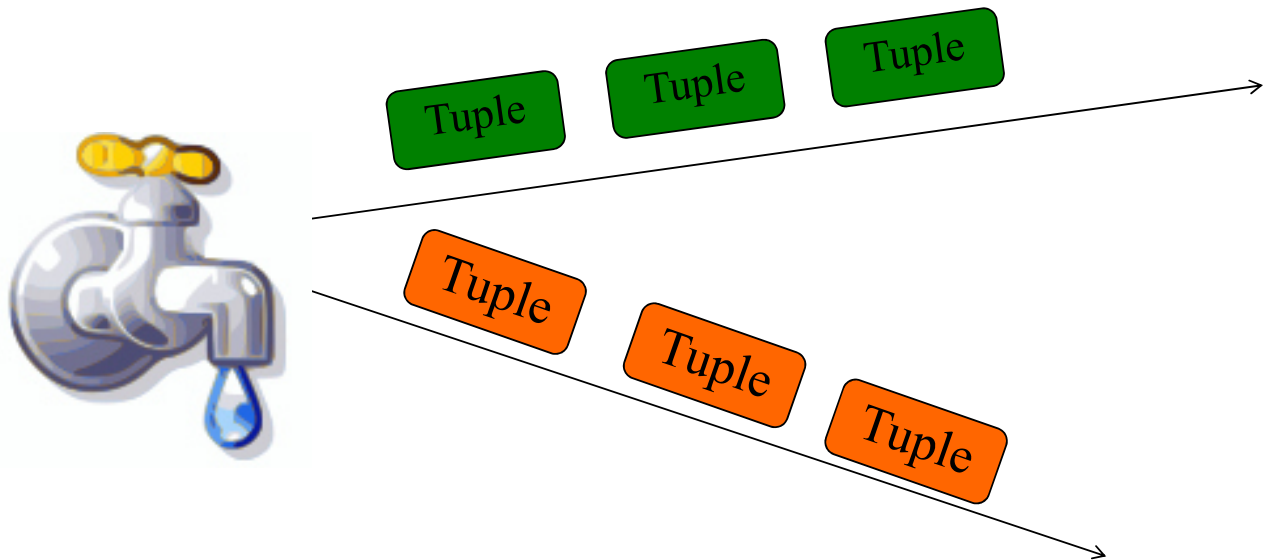
Tuples potentially unbounded in number

- Social network example:
 - <“Miley Cyrus”, “Hey! Here’s my new song!”>,
<“Justin Bieber”, “Hey! Here’s MY new song!”>,
<“Rolling Stones”, “Hey! Here’s my old song that’s still a super-hit!”>,
...
- Website example:
 - <coursera.org, 101.102.103.104, 4/4/2014, 10:35:40>,
<coursera.org, 101.102.103.105, 4/4/2014, 10:35:42>, ...

Spout

One **spout** is a **Storm entity** (process) that is a source of streams

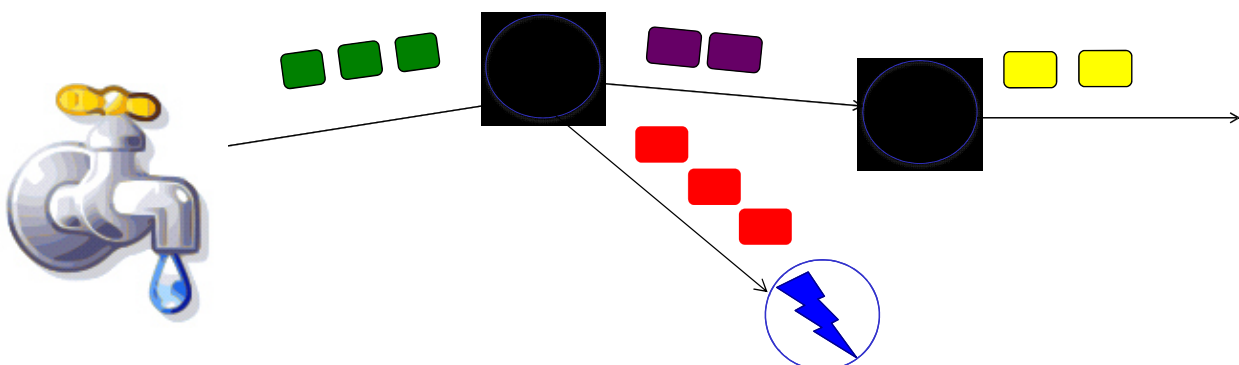
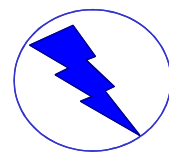
- Often reads from a crawler or DB



Bolt

A **bolt** is a **Storm entity** (process) that

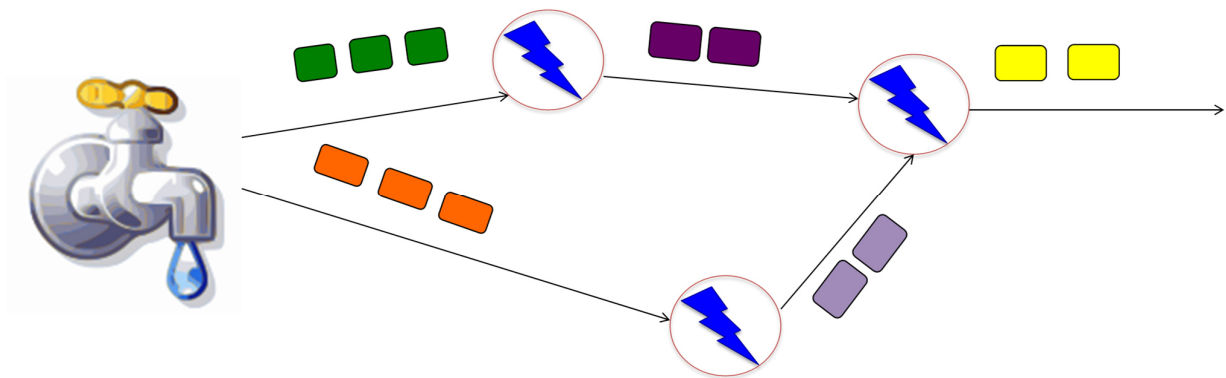
- Processes input streams
- Outputs more streams for other bolts



Topology

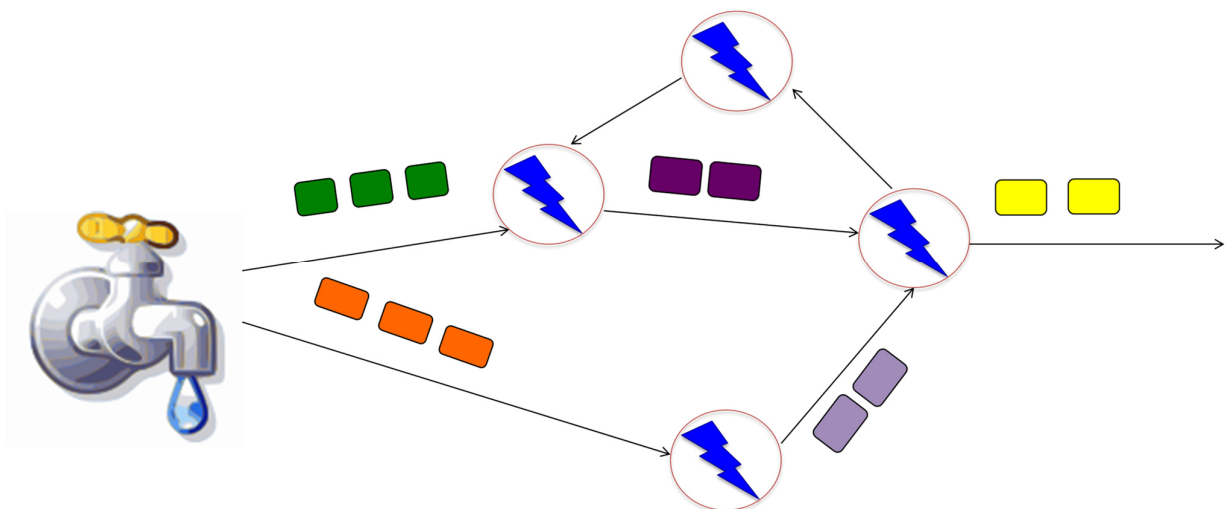
A **directed graph** of **spouts** and **bolts** (and output bolts)

- Corresponds to a Storm “application”



Topology

A **Storm topology** may define an architecture that can also have **cycles** if the application needs them



Bolts come in many Flavors

Operations that can be performed

- **Filter**: forward only tuples which satisfy a condition
- **Joins**: When receiving two streams A and B, output all pairs (A,B) which satisfy a condition
- **Apply/transform**: Modify each tuple according to a function
- ...And many others

But bolts need to process a lot of data

- Need to make them fast

Parallelizing Bolts

- **Storm** provides also **multiple processes (“tasks”)** that can constitute a bolt
- **Incoming streams** split among the tasks
- Typically each **incoming tuple goes to one task** in the bolt
 - Decided by “**Grouping strategy**”
- Three **types of grouping** are popular

Grouping

- **Shuffle Grouping**
 - Streams are distributed evenly among the bolt's tasks
 - Round-robin fashion
- **Fields Grouping**
 - Group a stream by a subset of its fields
 - E.g., All tweets where twitter username starts with [A-M,a-m,0-4] goes to task 1, and all tweets starting with [N-Z,n-z,5-9] go to task 2
- **All Grouping**
 - All tasks of bolt receive all input tuples
 - Useful for joins

Failures behaviour

Also failures can be mapped

- A tuple is considered failed when its topology (graph) of **resulting tuples fails to be fully processed within a specified timeout (time dimension)**
- **Anchoring**: Anchor an output to one or more input tuples
 - Failure of one tuple causes one or more tuples to be replayed

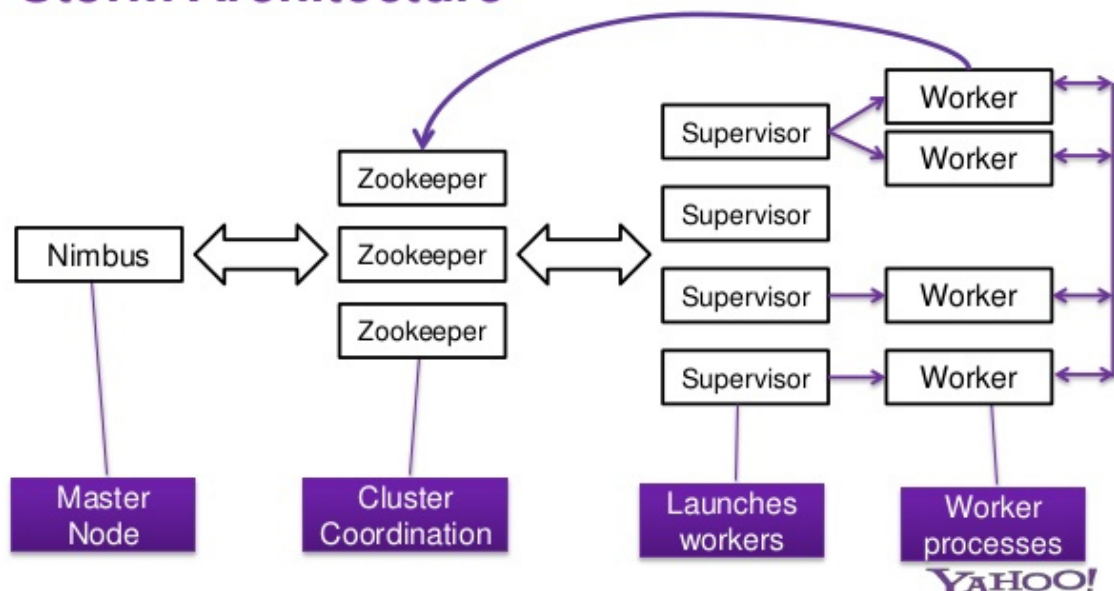
API For Fault-Tolerance (OutputCollector)

- **Emit** (tuple, output)
 - **Emits an output tuple**, perhaps anchored on an input tuple (first argument)
- **Ack** (tuple)
 - Acknowledge that a bolt **finished** processing a tuple
- **Fail** (tuple)
 - Immediately fail the spout tuple at the root of tuple topology if there is an exception from the database, etc.
- **Must Record the ack/fail of each tuple**
 - Each tuple consumes memory. Failure to do so results in memory leaks.

Storm Cluster

Several components in a Cluster

Storm Architecture



Zookeeper

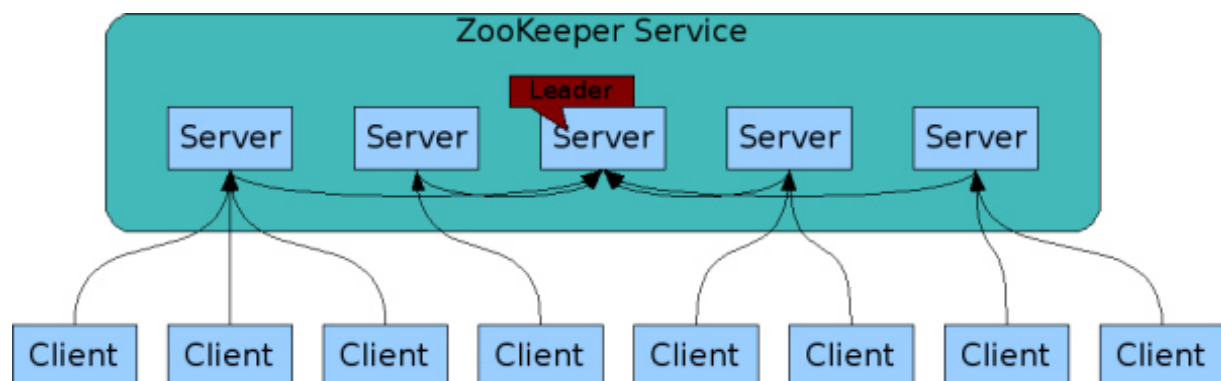
ZooKeeper is an open-source Distributed Coordination Service for Distributed Applications

- ZooKeeper can propose a unique **memory space to be accessed very fast in reading and writing with quality** (QoS: replication is paramount and dynamicity too)
- ZooKeeper relieves distributed applications from implementing coordination services from scratch
- ZooKeeper exposes a simple set of primitives to implement higher level services for synchronization, configuration maintenance, and groups and naming
- It is easy to program and uses a data model styled after the familiar directory tree structure of file systems. It runs in Java and has bindings for both Java and C

Zookeeper

ZooKeeper is seen as a unique access space with a very fast operations to read and write data with different semantics (FIFO, Atomic, Causal, ...)

Data are dynamically mapped over several nodes and their location can be dynamically adjusted without any actions of clients



Storm Architecture

Storm allows to:

1. First express your need **in streaming via its components** you can easily define and design
2. Secondly, configure your **capacity needs over a real architecture** so to produce a controlled execution
3. Then **operate it over different architectures**

Storm Cluster

- **Master node**
 - Runs a daemon called *Nimbus*
 - Responsible for
 - ✓ Distributing code around cluster
 - ✓ Assigning tasks to machines
 - ✓ Monitoring for failures of machines
- **Worker node**
 - Runs on a machine (server)
 - Runs a daemon called *Supervisor*
 - Listens for work assigned to its machines
 - Runs “Executors”(which contain groups of tasks)
- **Zookeeper**
 - Coordinates Nimbus and Supervisors communication
 - All state of Supervisor and Nimbus is kept here

Twitter Heron System

Fixes the inefficiencies of Storm acknowledgement mechanism (among other things)

By using **backpressure**: a **congested downstream tuple** will ask upstream tuples to slow or stop sending tuples

1. **TCP Backpressure**: uses TCP windowing mechanism to propagate backpressure
 2. **Spout Backpressure**: node stops reading from its upstream spouts
 3. **Stage by Stage Backpressure**: think of the topology as stage-based, and propagate back via stages
- By using:
 - Spout+TCP, or
 - Stage by Stage + TCP
 - Heron beats Storm throughput

S4 Platform

Simple Scalable Streaming System (S4)

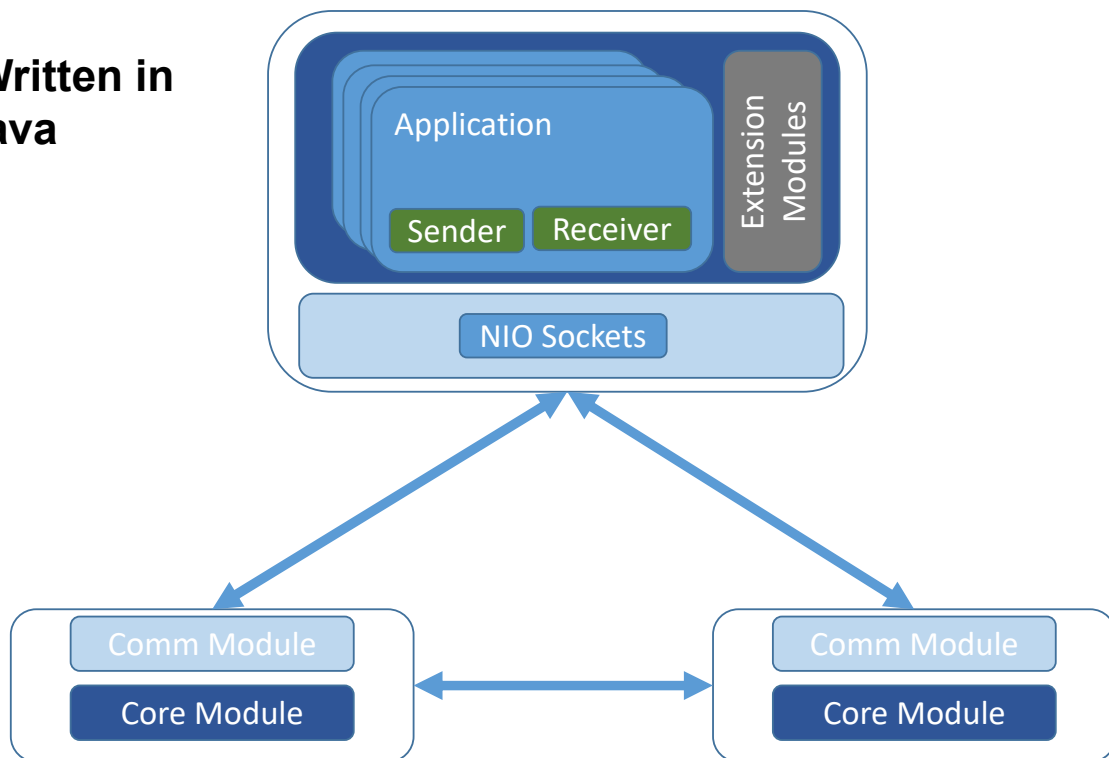
S4 is a **general-purpose, near real-time, distributed, decentralized, scalable, event-driven, modular platform** that allows to implement applications for processing **continuous unbounded streams of data**

Design goals:

- **Scalability**
- **Decentralization**
- **Fault-tolerance (partially supported)**
- **Elasticity**
- **Extensibility**
- **Object oriented**

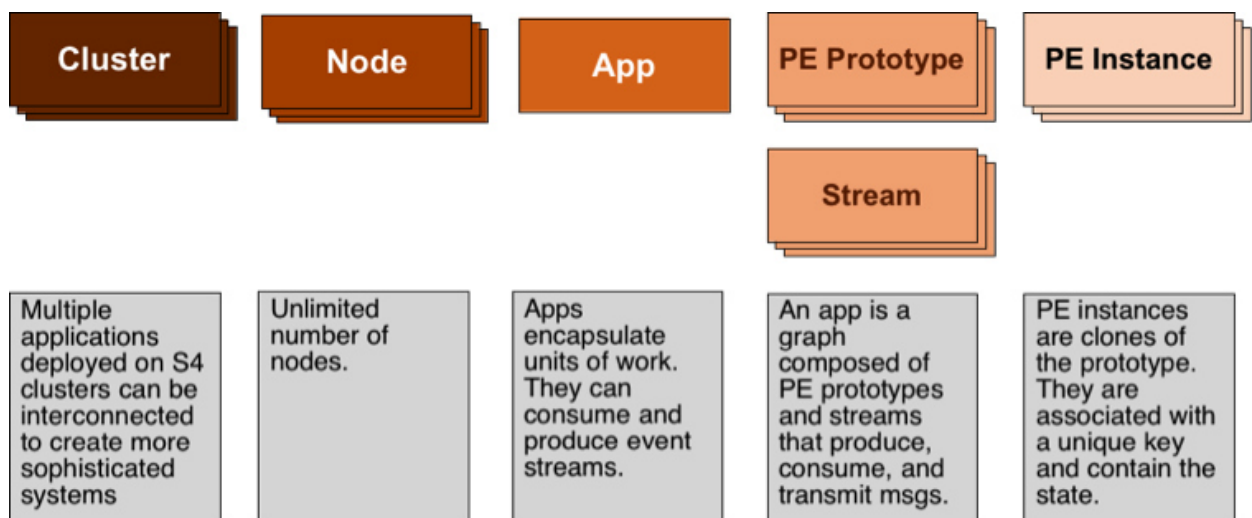
S4 Platform - architecture

Written in
java

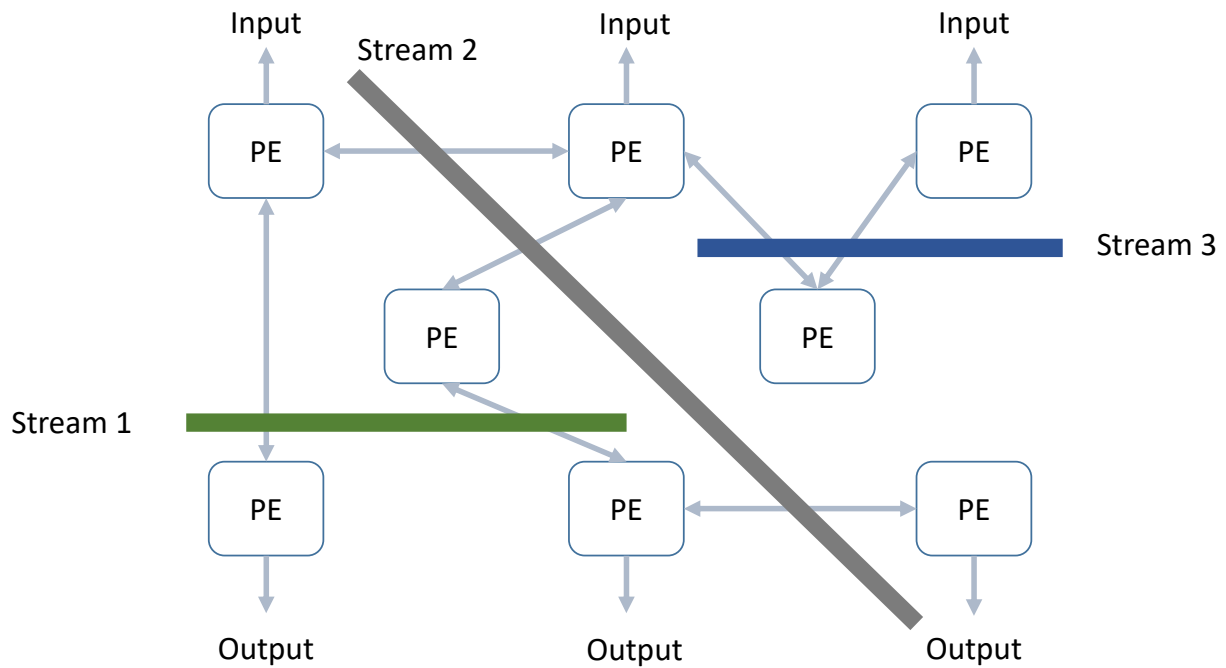


S4 Platform - components

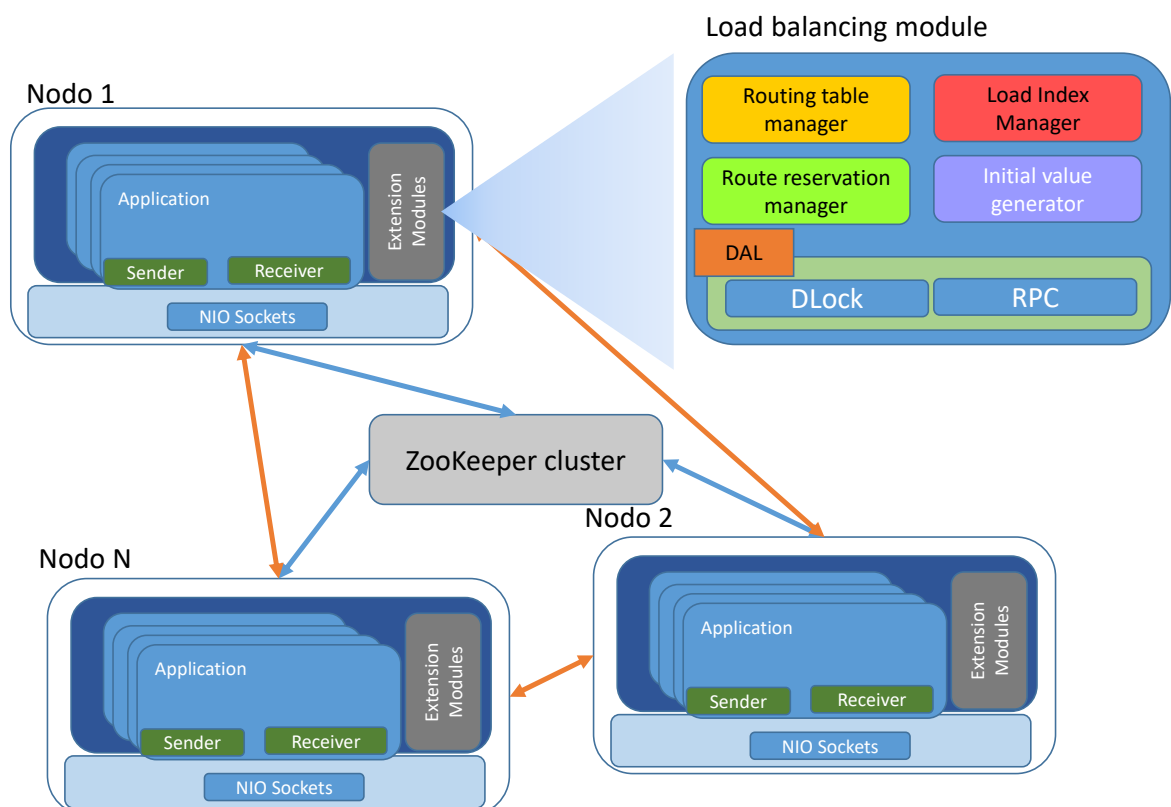
S4 based on several simple components that can be put together



S4 Platform - application



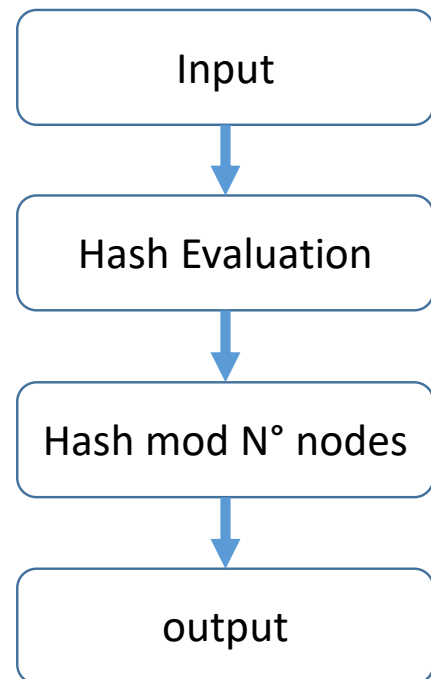
S4 Platform – overall view



Load balancing support & open issues

Not really supported...

- **Load sharing on cluster nodes based on very simple hash functions**
- **There is no real load balancing support**
- **No guarantees of effectively balanced load sharding**



An example: Word Count (sounds familiar?)

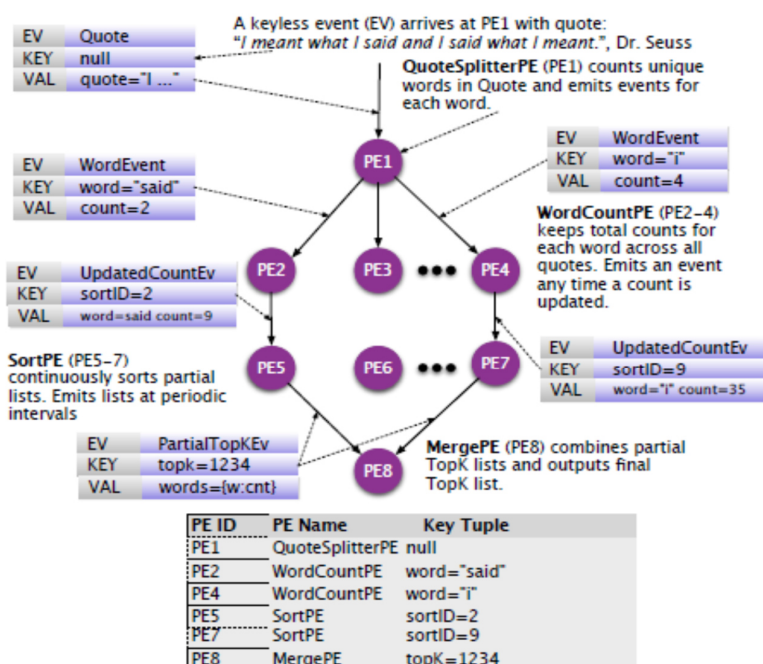


Figure 1. Word Count Example

For details, refer to the S4 presentation paper: L. Neumeyer *et al.*, "S4: Distributed Stream Computing Platform", KDCloud 2010