



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

Class of
**Infrastructures for
Cloud Computing and Big Data M**

Global Stream Processing

Antonio Corradi
Academic year 2020/2021

STREAM PROCESSING

There is more and more interest on **stream processing** ...
so **Automatize everything** – for **special-purpose** behavior

We mean that you must process data not defined in
dimension but with a contract of (possibly)

unbounded scale not known at all

Keep in mind the **importance of such streamings**

More and more **sets of tools** become 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 contents of data with some time constraints

- **with latencies of few seconds**
- **with high throughput**

No way of using databases for storing

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 batch approaches are not intended for **long-running unbounded stream-processing**

But using message passing dataflows

MOM-like functions as connector among components

NOT MAPREDUCE

Message passing dataflow - asynchronous

MOM-like functions as connector among components

Problems:

Mismatch **velocities of output and input**

Crash of a **node**

Dynamic intervention on line

NOT MAPREDUCE

Message passing dataflow - asynchronous

MOM-like functions as connector among components

Problems:

Mismatch velocities of output and input

Buffering in a queue → cost / overflow

Crash of a node

Using replication and multiple copies

Dynamic intervention on line

Change graph while provisioning

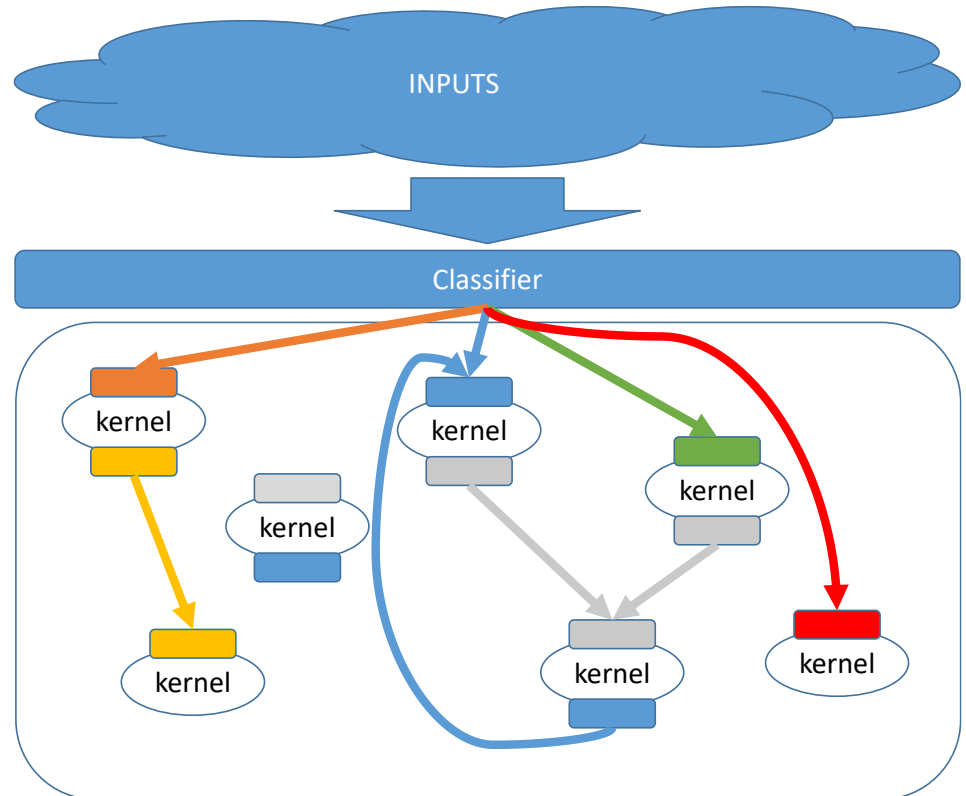
STREAM PROCESSING MODEL

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

Storm is fast in **processing over a million data 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

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.

Core Components:

Tuples, Streams, Spouts, Bolts, Topologies

TUPLE

We have already seen **tuple** as a **set of values according to some attributes**

The tuple is an ordered list of elements

E.g., <tweeter, tweet>

- E.g., <“Miley Cyrus”, “Hey! Here’s my new song!”>
- E.g., <“Justin Bieber”, “Hey! Here’s MY new song!”>

Tuple

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:

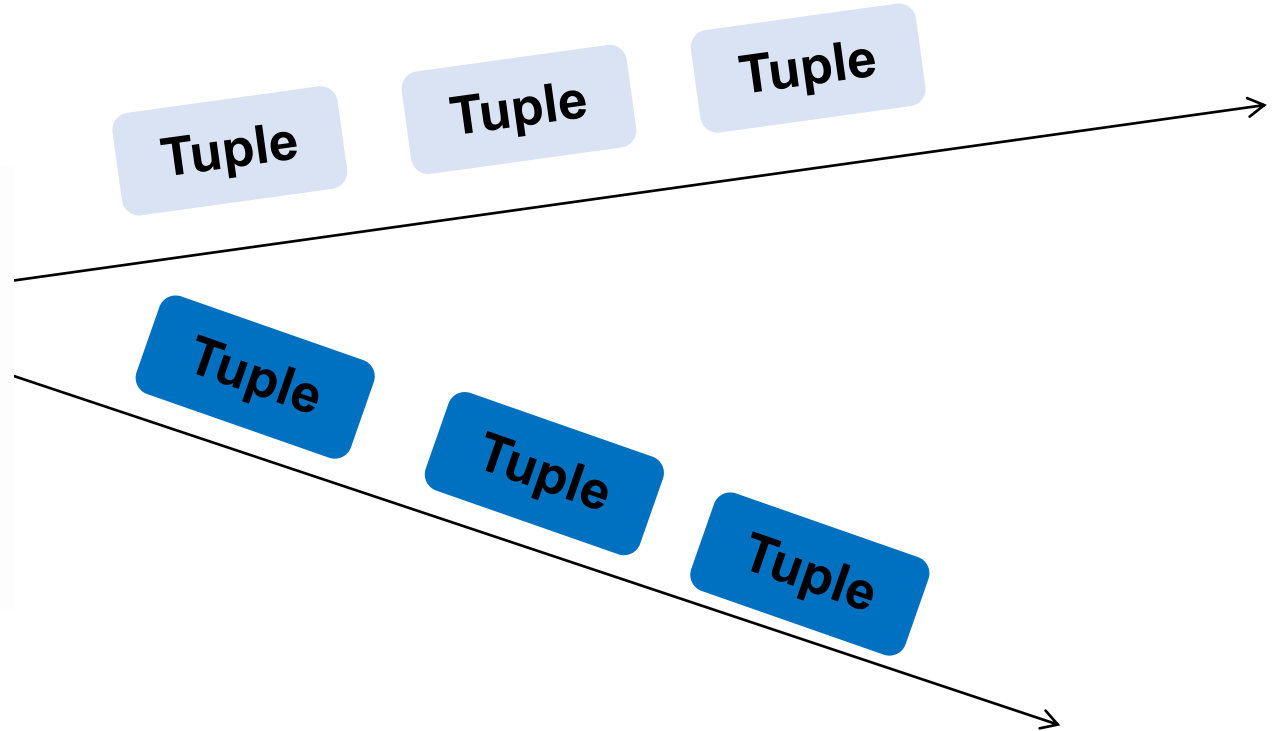
<coursera.org, 101.102.103.104, 4/4/2014, 10:35:40> ,

<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** (set of tuples)

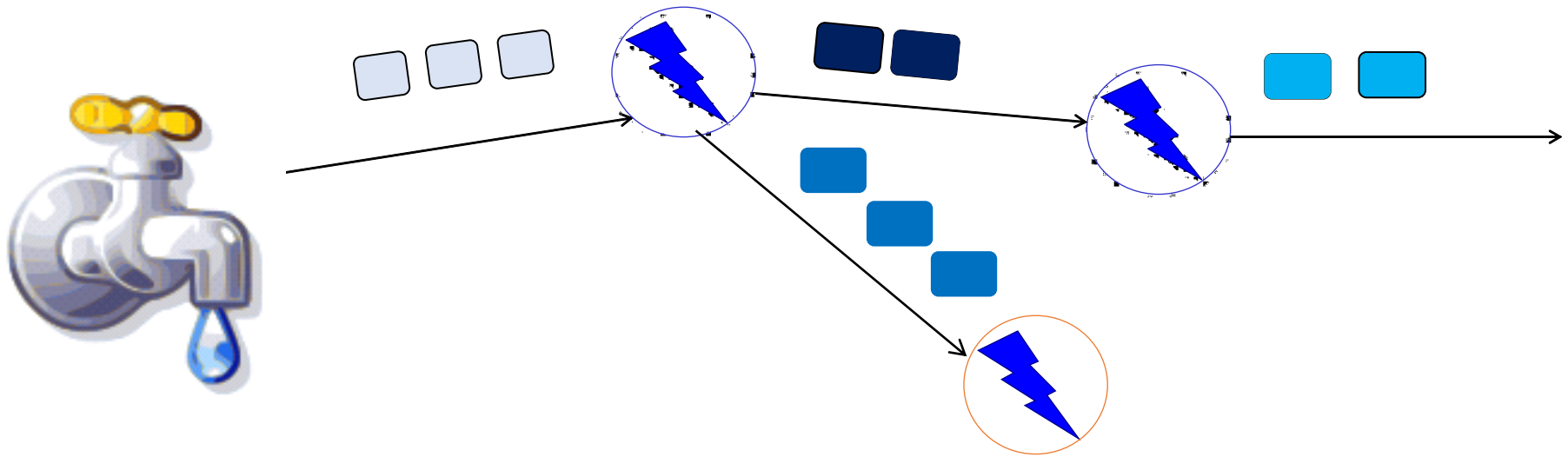
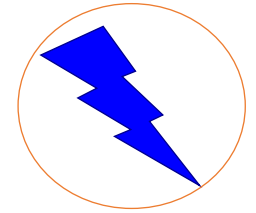
Often reads from a crawler or DB



SPOUT

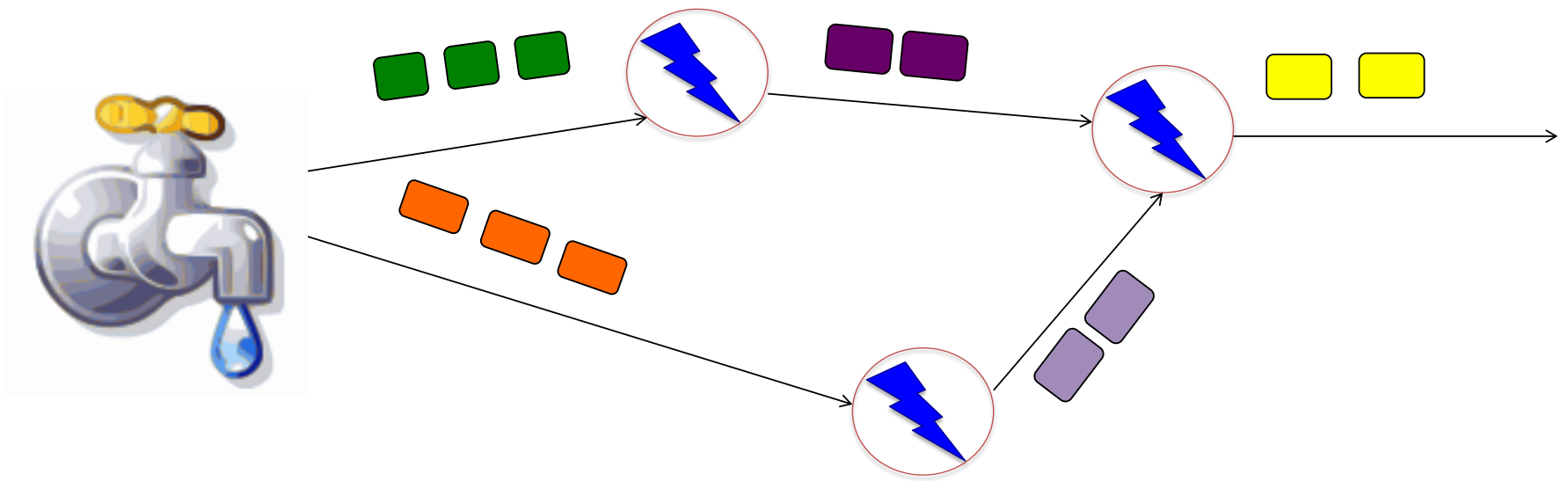
A **bolt** is a **Storm entity** (process) that can

- **process** input streams
- **output** more streams for other bolts



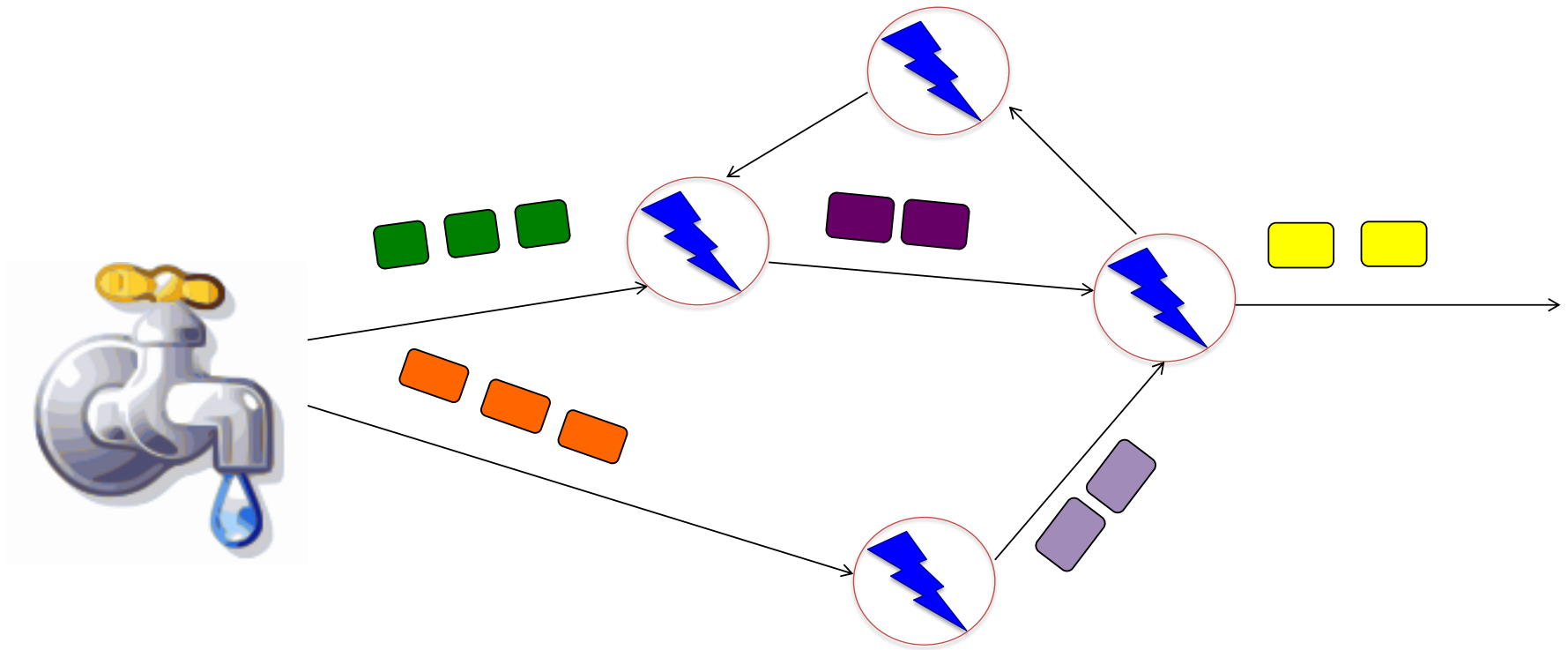
TOPOLOGY

A **directed graph** of **spouts** and **bolts** (and output bolts)
Corresponds to a **Storm “application”**



TOPOLOGY

A **Storm topology** may define a **storm architecture** that can also present **cycles** if the application needs them



BOLTS COME IN MANY FLAVORS

Bolts must be capable of operating over the data via tasks

Operations that can be performed by tasks

- **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 can be split among the tasks

Typically each **incoming tuple goes to one task** in the bolt

- Decided by “**Grouping strategy**”

GROUPING

Three **types of grouping** are popular

Shuffle Grouping

- Streams are distributed evenly among the bolt tasks
- Round-robin fashion

Fields Grouping

Group a stream by a subset of its fields such as

- 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

FAILURE BEHAVIOR

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.

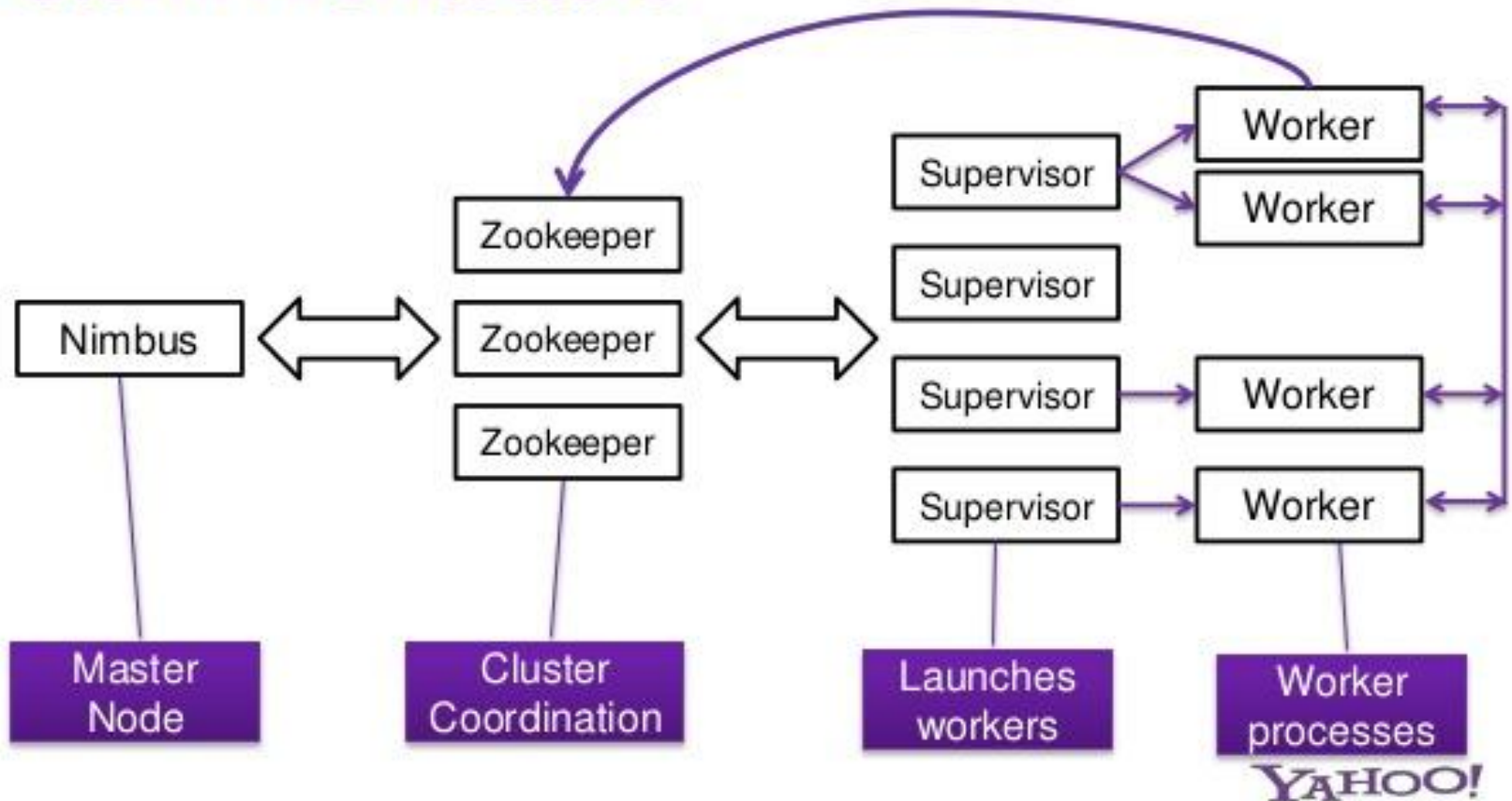
Must Record the **ack/fail** of each tuple

- 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**:

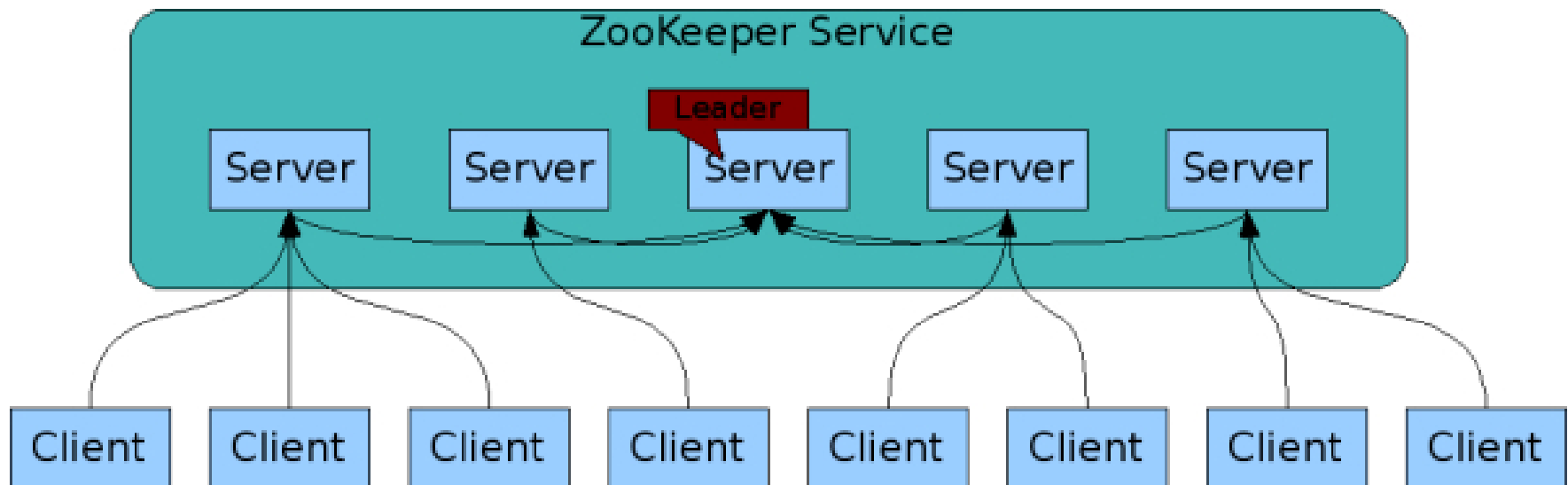
- can propose a unique **memory space with very fast access in reading and writing** with **some quality** (QoS: replication is paramount and dynamicity too)
- relieves distributed applications from **implementing coordination services from scratch**
- exposes a **simple set of primitives** to implement **higher level services for synchronization, configuration maintenance, and groups and naming**

The data model is shaped after the familiar **directory tree structure of file systems** and it runs in Java with bindings for both Java and C

ZOOKEEPER

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

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

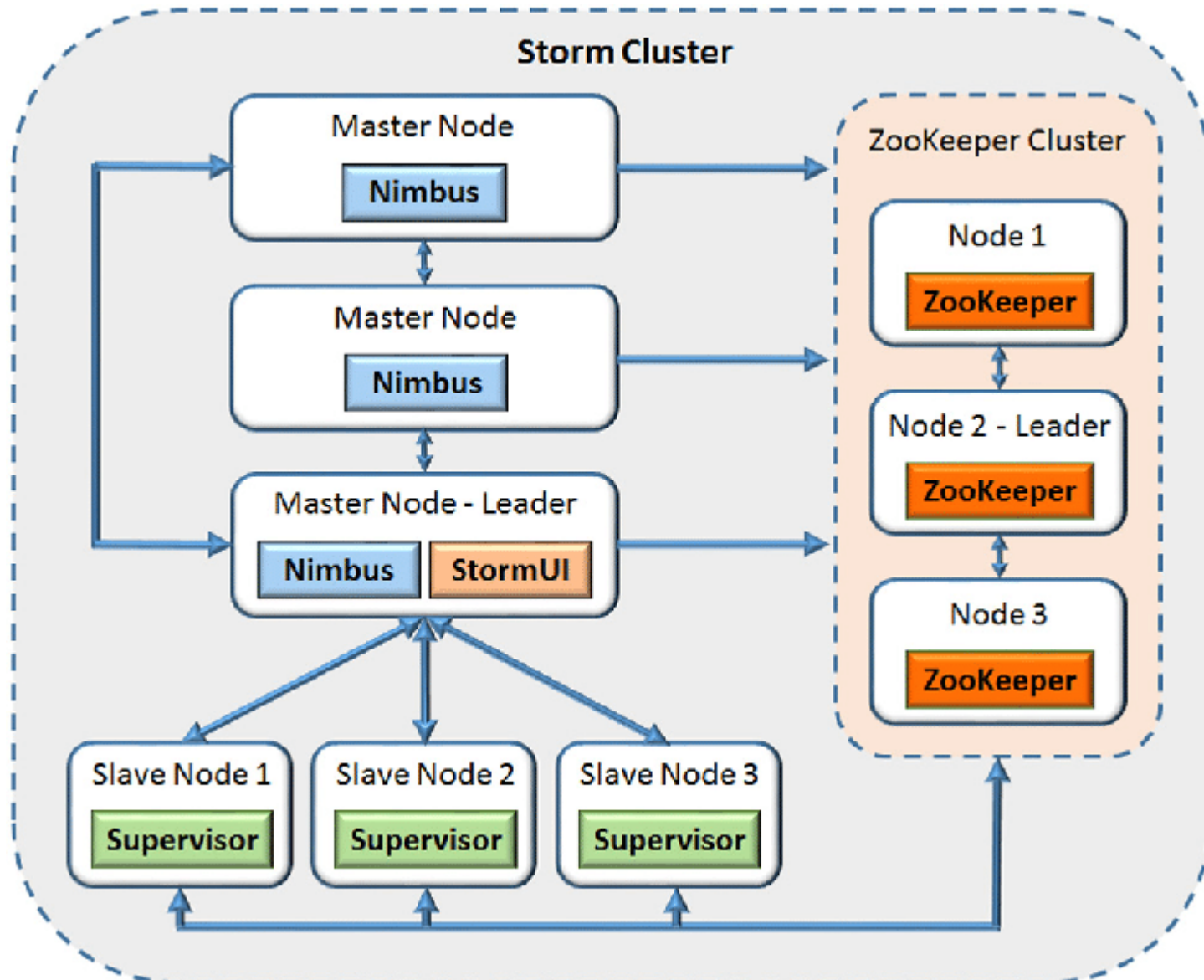


STORM ARCHITECTURE

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 ARCHITECTURE



STORM CLUSTER

Master node

- Runs a daemon called *Nimbus*
- Responsible for
 - ✓ Distributing code around cluster
 - ✓ Assigning tasks to machines
 - ✓ Monitoring for failures of machines

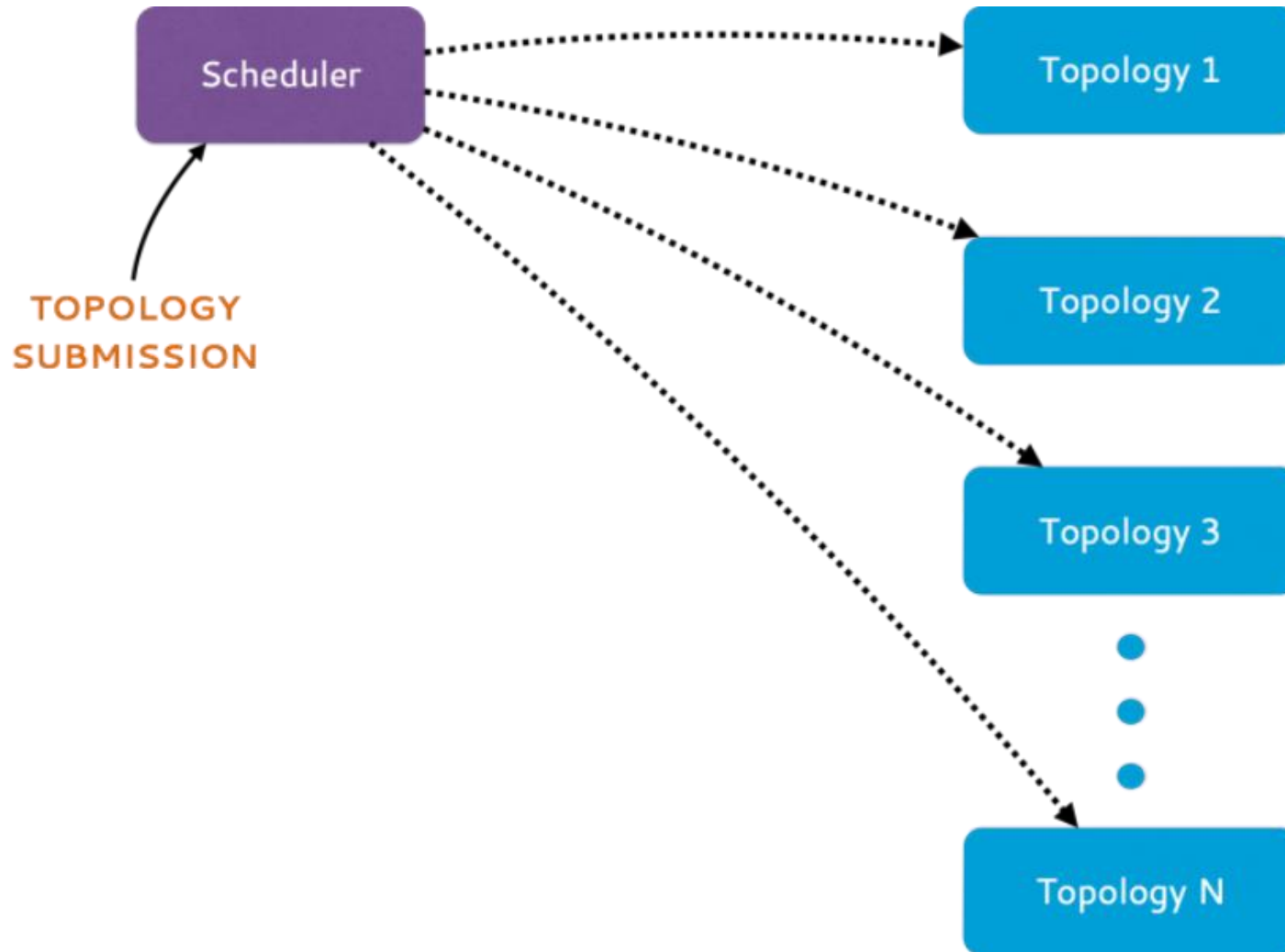
Worker node

- 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

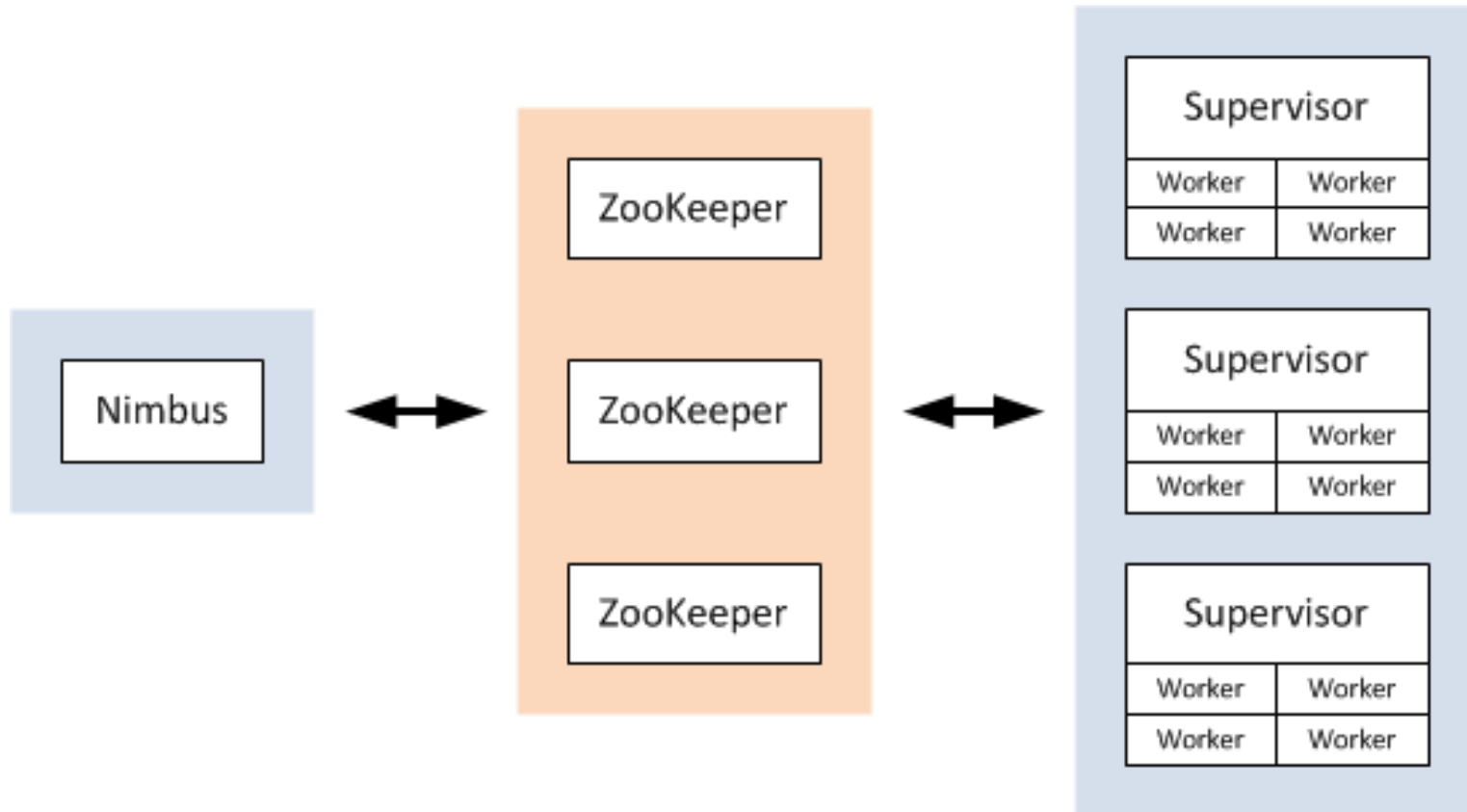
- Coordinates Nimbus and Supervisors communication
- All state of Supervisor and Nimbus is kept here

TWITTER HERON SYSTEM

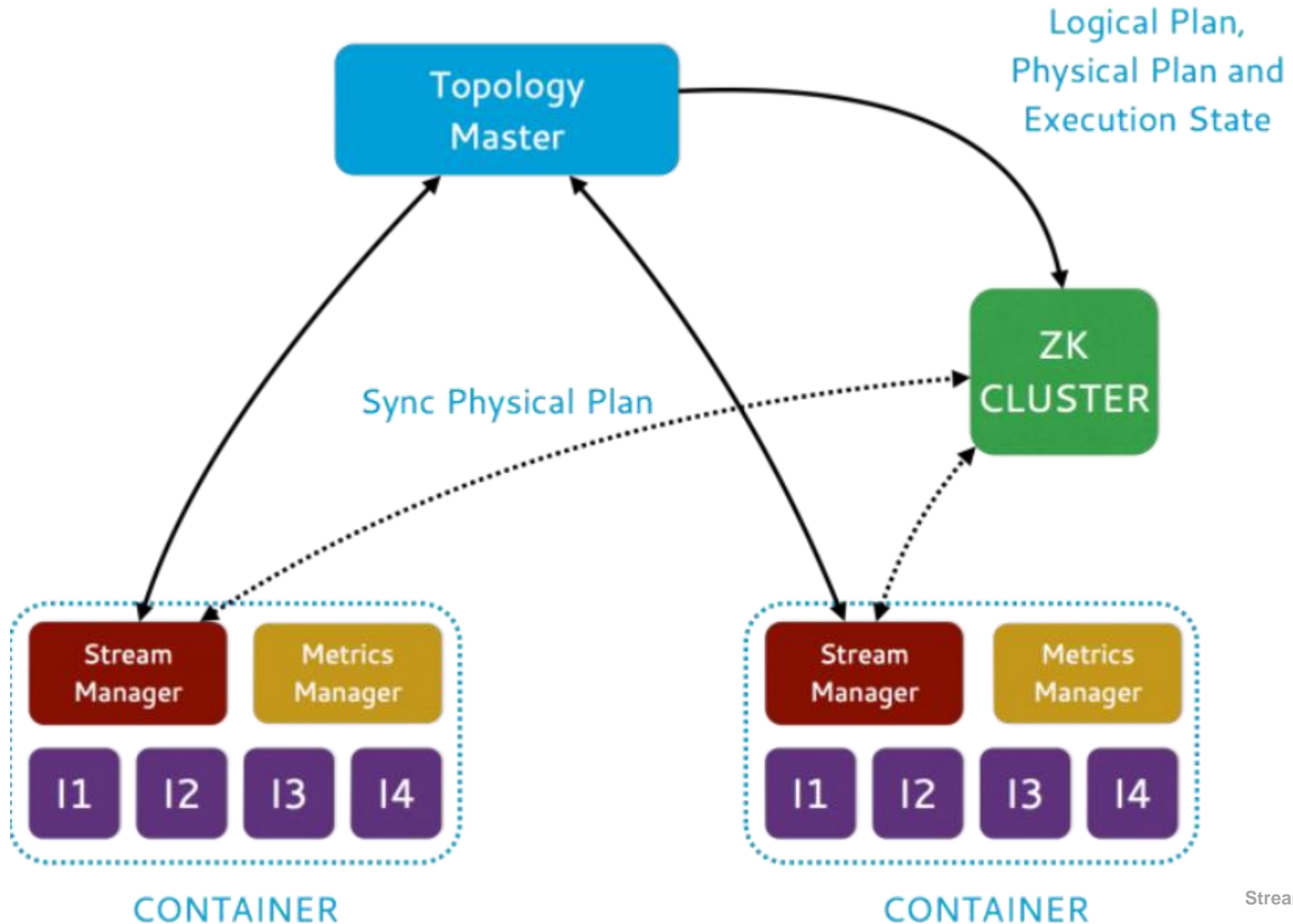


STORM MONITORING SYSTEM

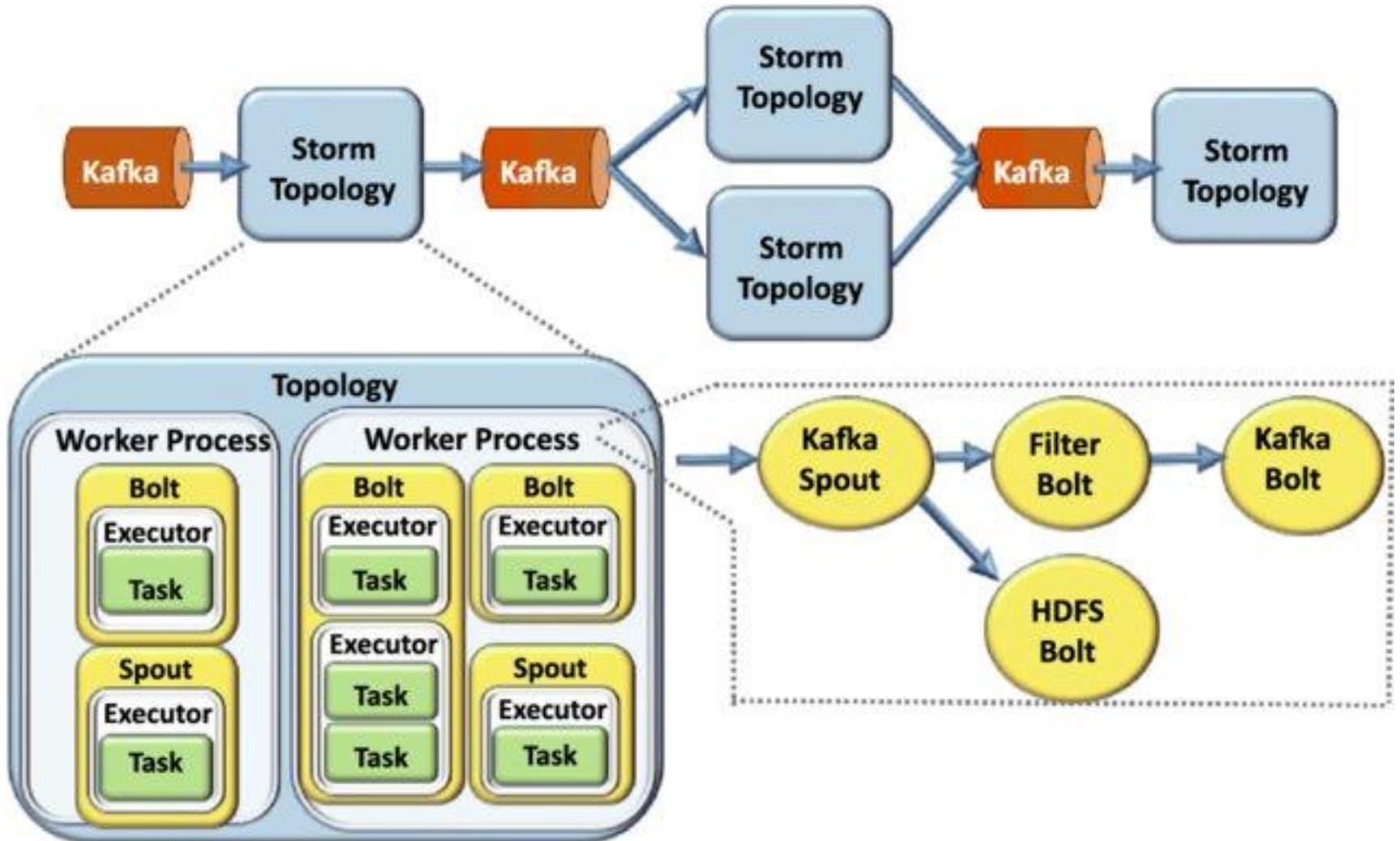
Monitoring and Control in Storm



TWITTER HERON SYSTEM



MAPPING OF STORM



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

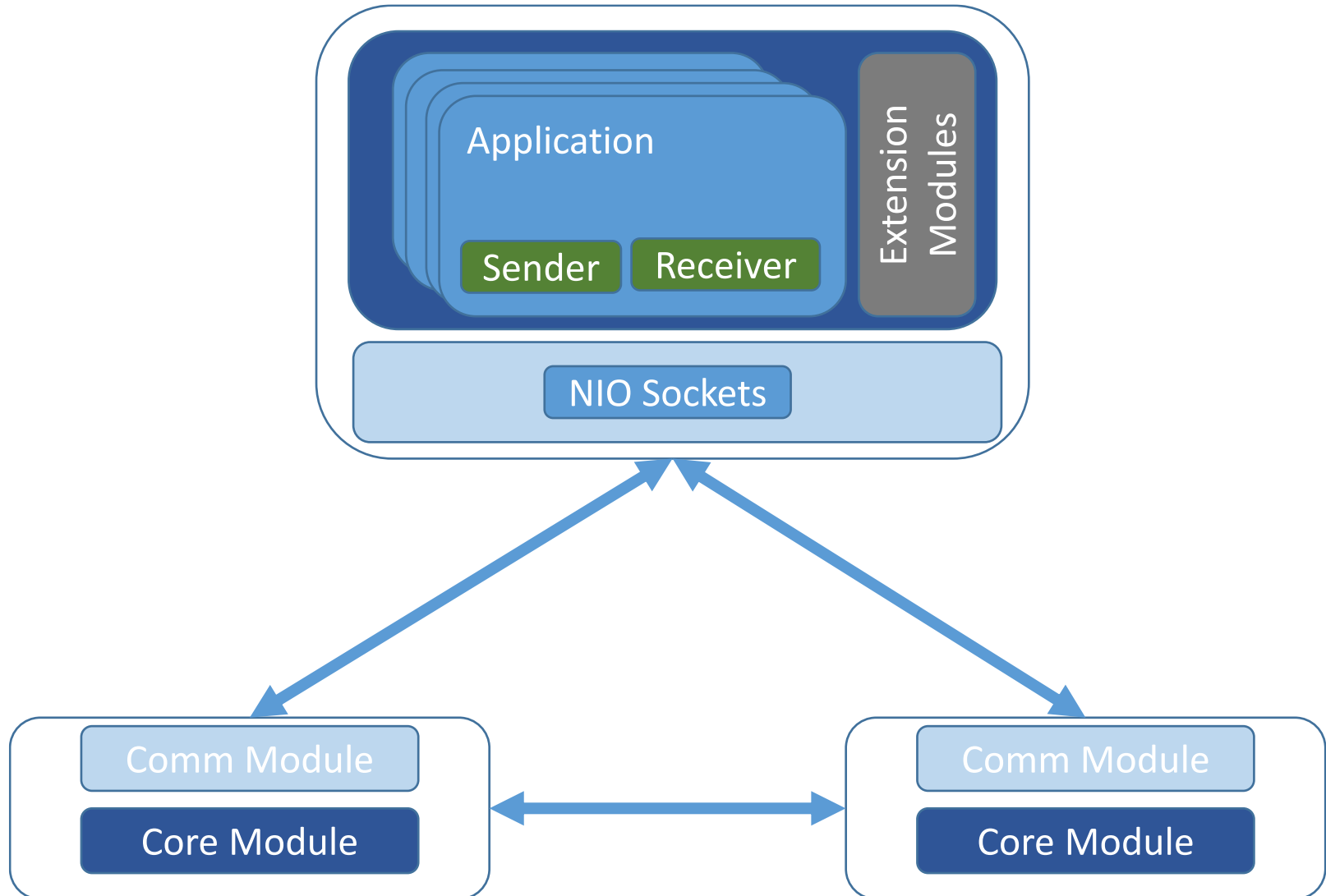
Simple Scalable Streaming System (S4)

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:

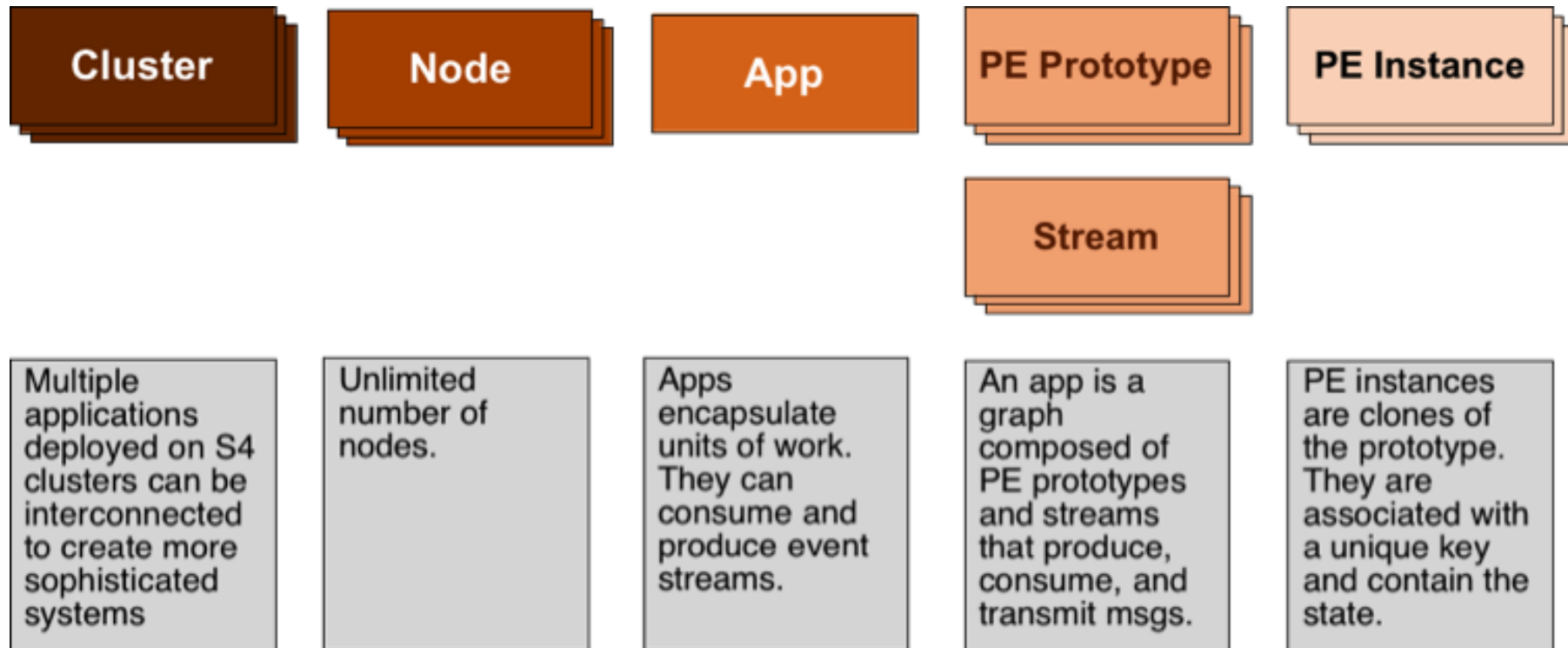
- **Scalability**
- **Decentralization**
- **Fault-tolerance (partially supported)**
- **Elasticity**
- **Extensibility**
- **Object oriented**

S4 PLATFORM - ARCHITECTURE

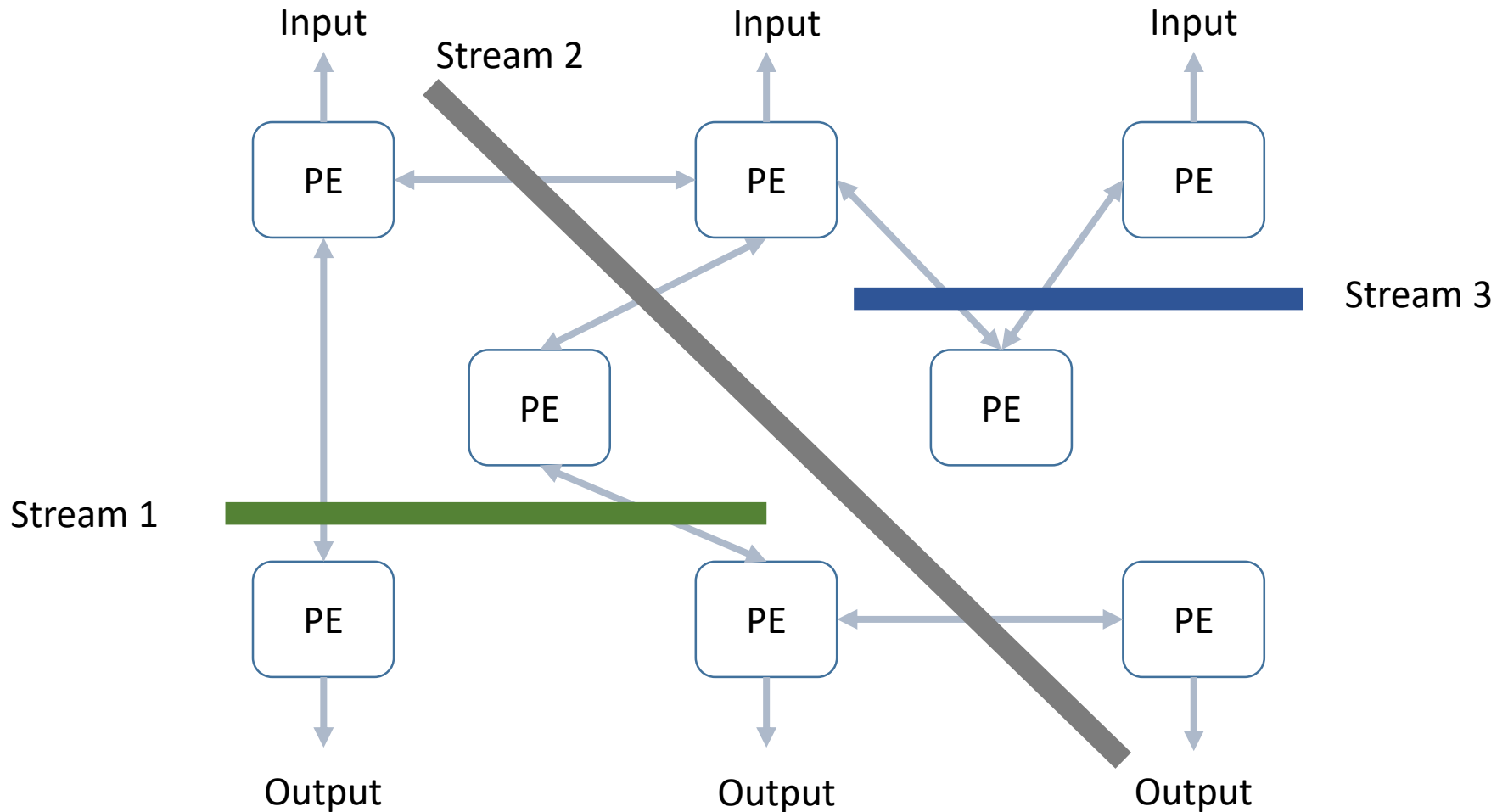


S4 PLATFORM - COMPONENTS

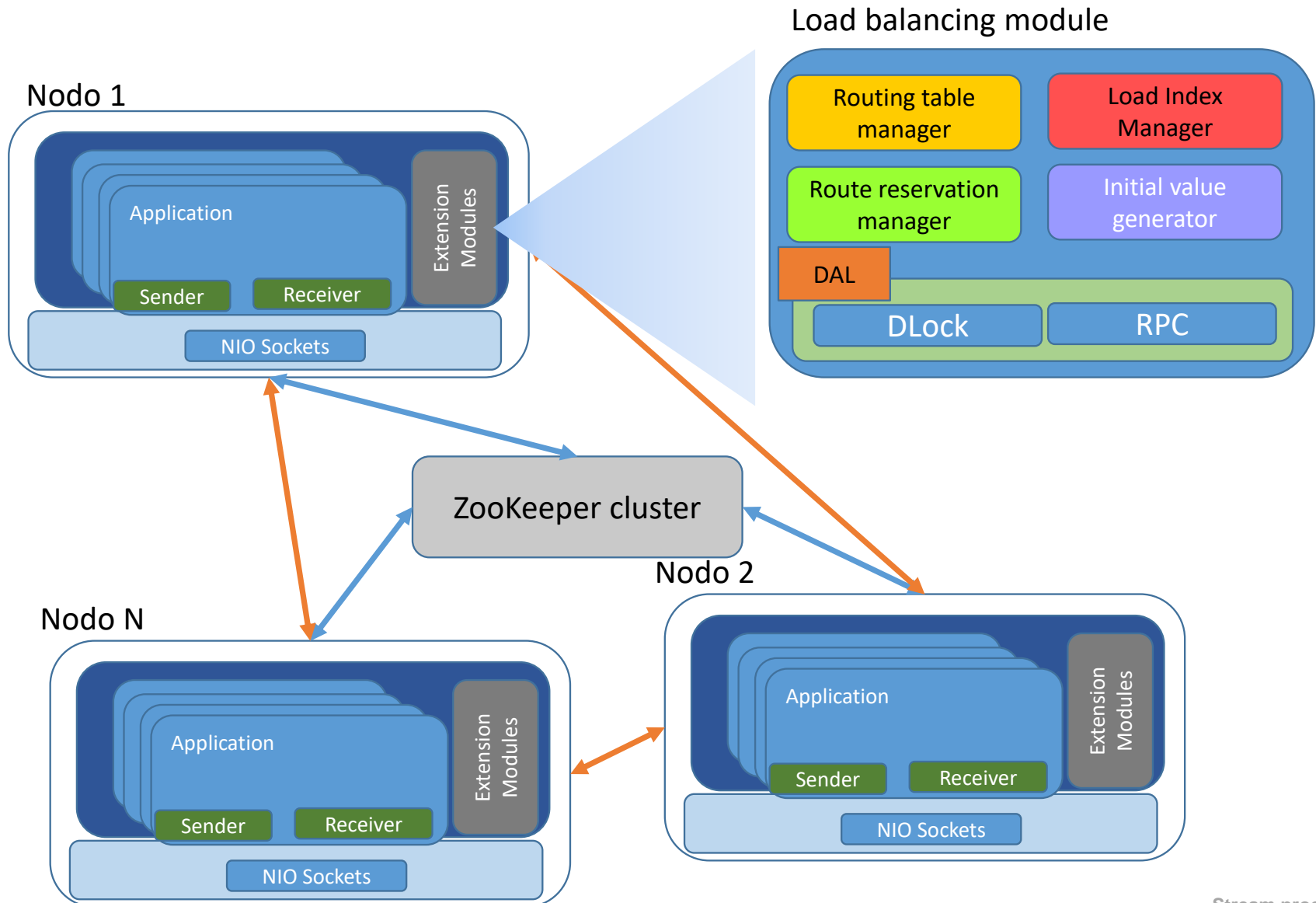
S4 based on several simple components that can be put together



S4 PLATFORM - APPLICATIONS



S4 PLATFORM – OVERALL VIEW



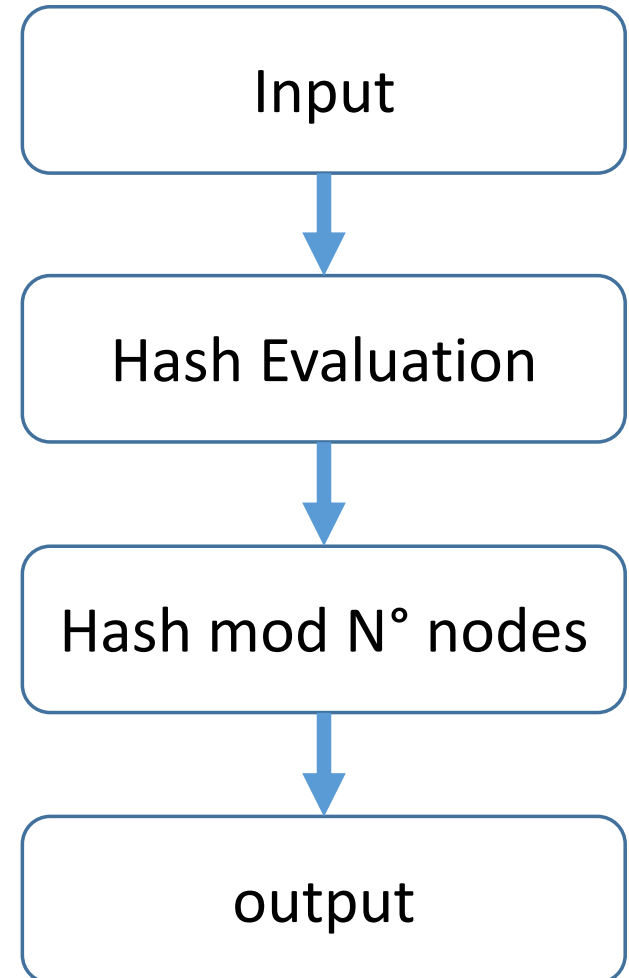
LOAD BALANCING SUPPORT & OPEN ISSUES

Not really supported...

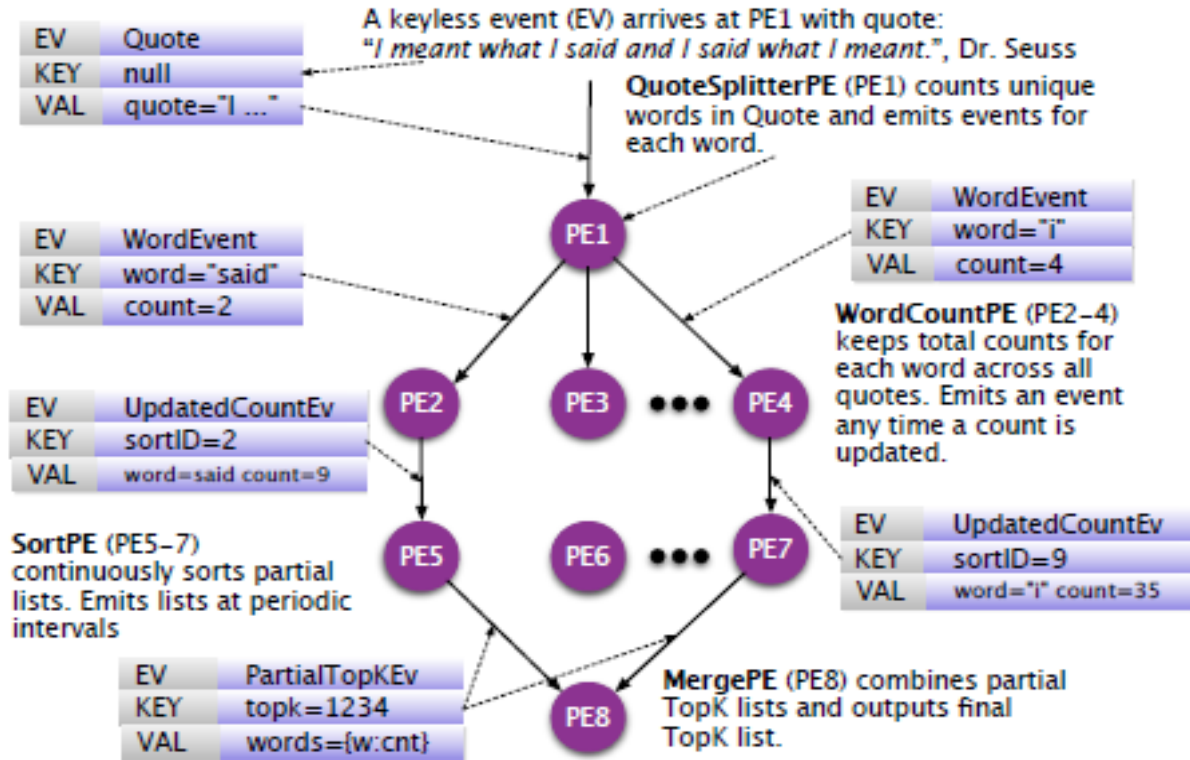
Load sharing on cluster nodes based on very simple hash functions

There is no real load balancing support

No guarantees of effectively balanced load sharding



AN EXAMPLE: WORD COUNT (SOUNDS FAMILIAR?)



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

PE ID	PE Name	Key Tuple
PE1	QuoteSplitterPE	null
PE2	WordCountPE	word="said"
PE4	WordCountPE	word="I"
PE5	SortPE	sortID=2
PE7	SortPE	sortID=9
PE8	MergePE	topK=1234

Figure 1. Word Count Example