QCON Beijing 2015

Gearpump
Realtime Streaming on Akka

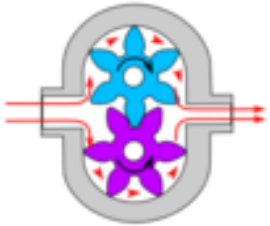
Sean Zhong
Mail: xiang.zhong@intel.com
Weibo：http://weibo.com/clockfly

Intel SSG
Big Data Technology Department

2015/4/24

What is Gearpump 

- **Akka** based lightweight **Real time data processing** platform.
- Apache License http://gearpump.io



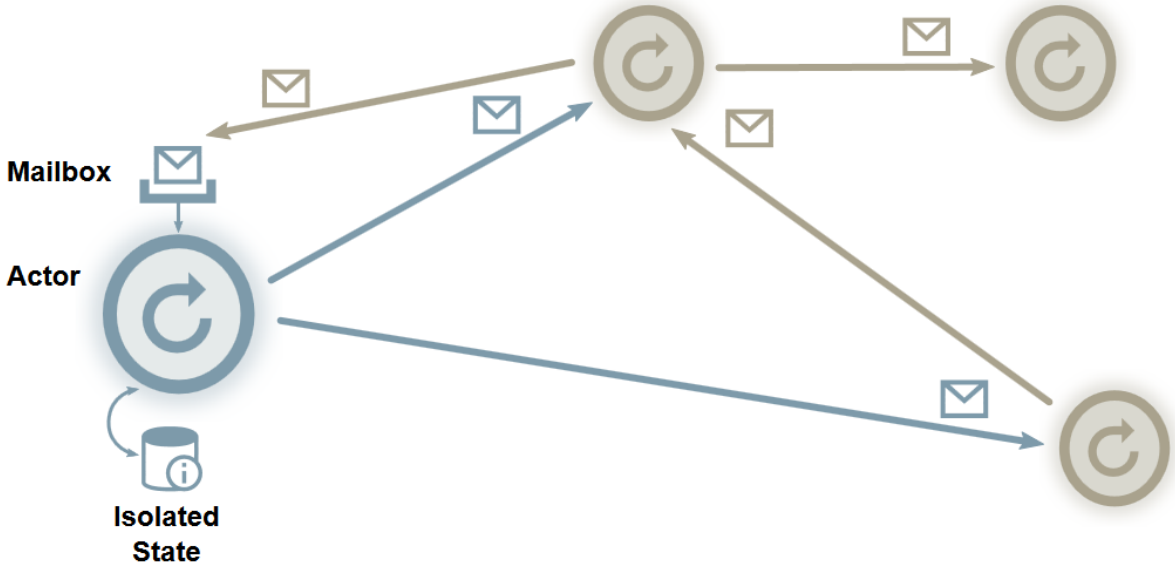
Simple and **Powerful**
Message level streaming
Long running daemons

- **Akka:**
- **Communication, concurrency, Isolation, and fault-tolerant**

What is Akka?

What is Akka?

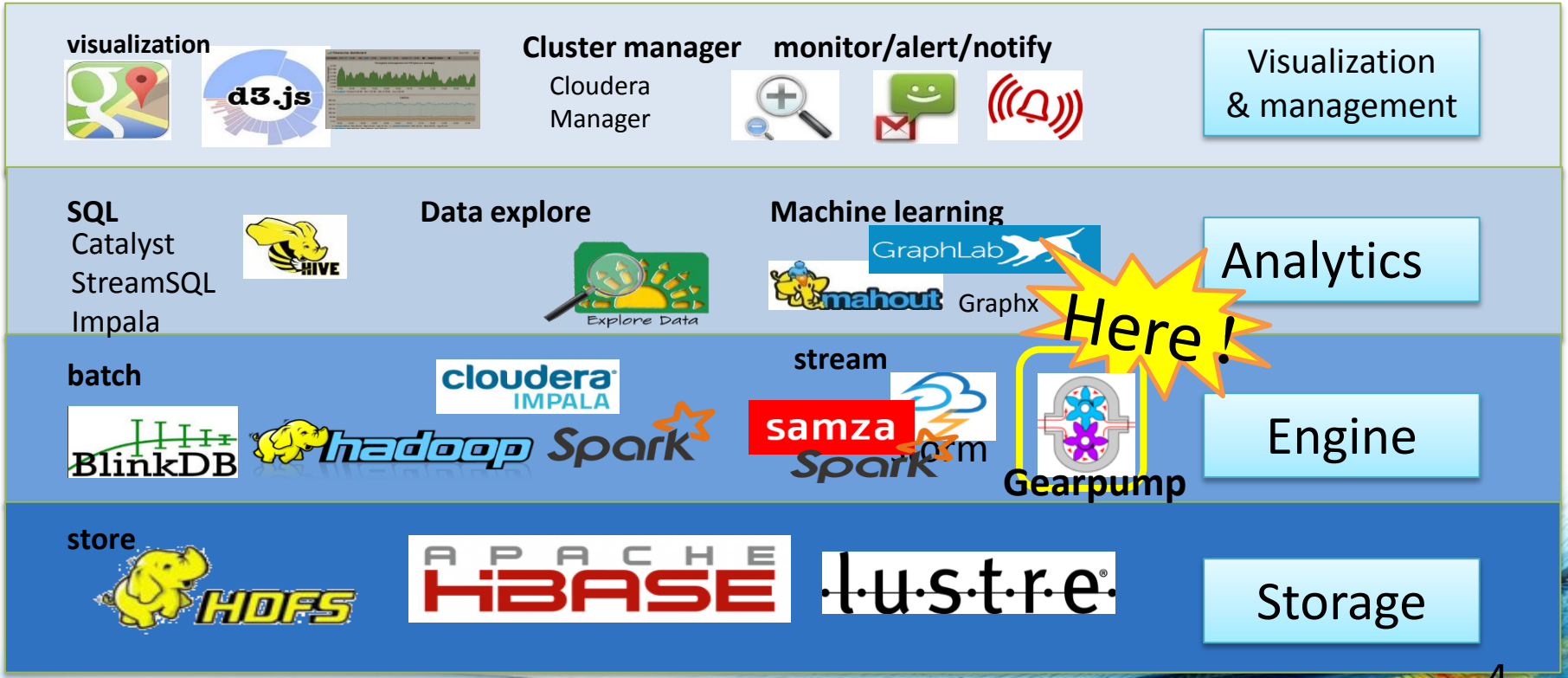
- Micro-service(Actor) oriented.



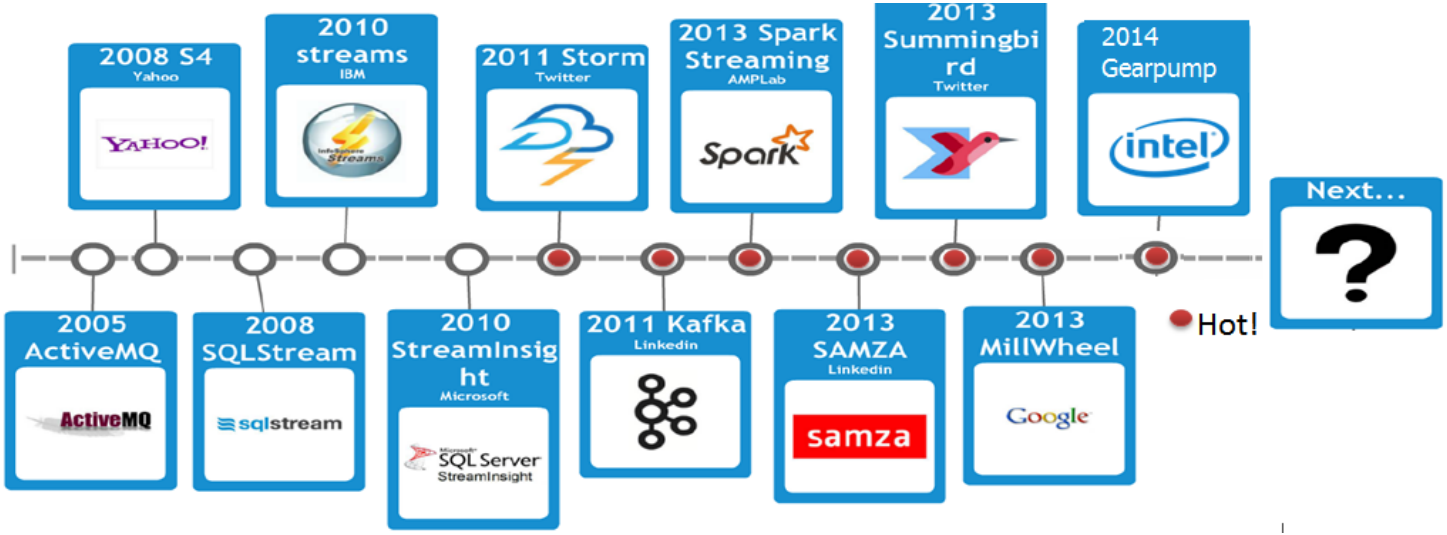
- **Message Driven**
- **Lock-free**
- **Location-transparent**
- **Better performance**
- Fail in-dependently
- Scales linearly

It is like our human society, driven by message

Gearpump in Big Data Stack



Why another streaming platform?



- The **requirements** are not fully met.
- We need a **higher abstraction** to build software**!**

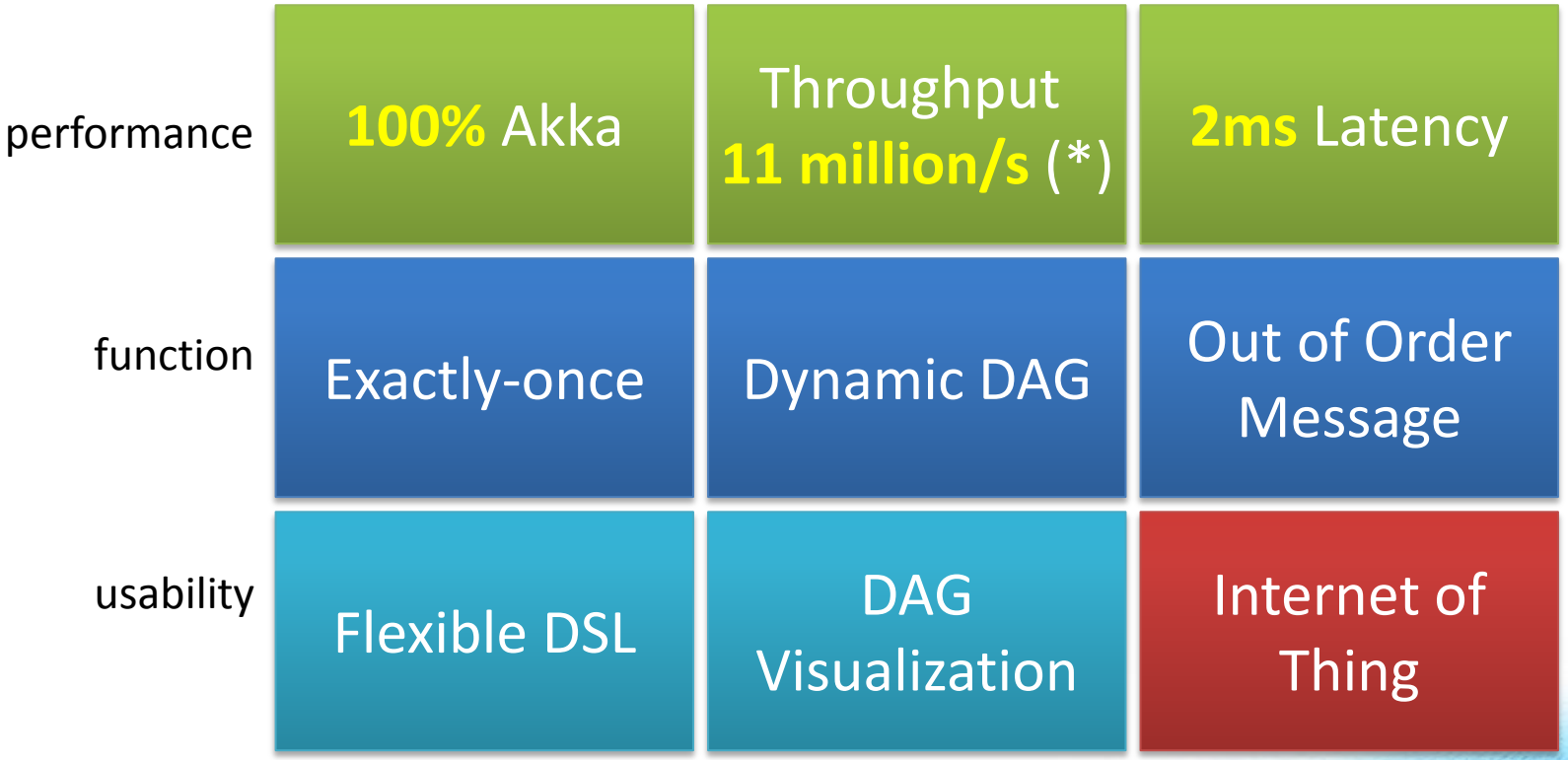
Streaming requirements

- Michael Stonebraker, *The **8** Requirements of Real-Time Stream Processing (2006)*

My summary

Flexible	Volume	Speed	**Ac**curacy	Visual
Easy programing Any time Any where Any size Any source Any use case Dynamic DAG **②StreamSQL**	High throughput ⑦Scale linearly	①In-Stream Zero latency ⑥HA ⑧Responsive	Exactly-once ③Message loss/delay/out of order ④Predictable	WYSWYG

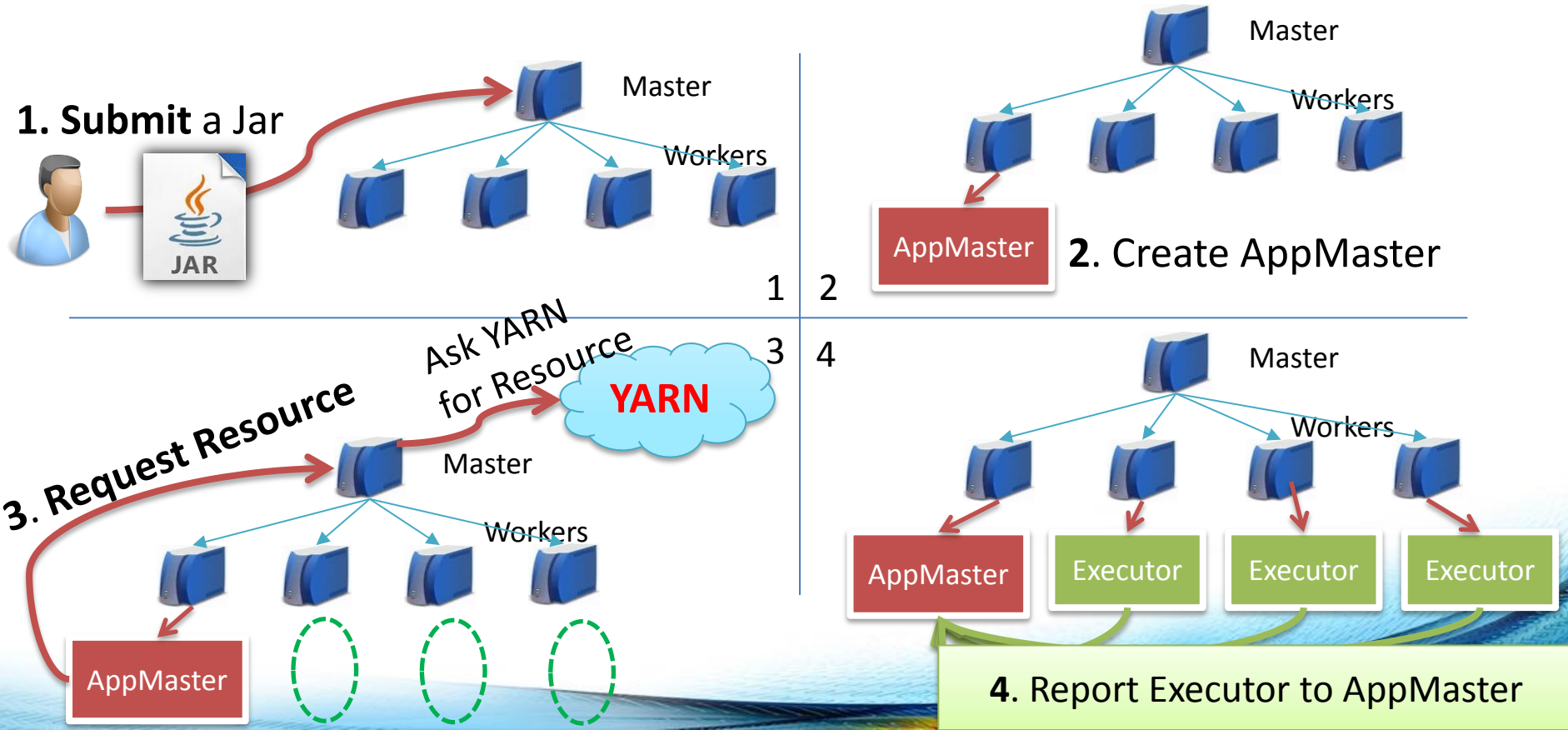
Gearpump Highlights



[*] on 4 nodes

USING GEARPUMP

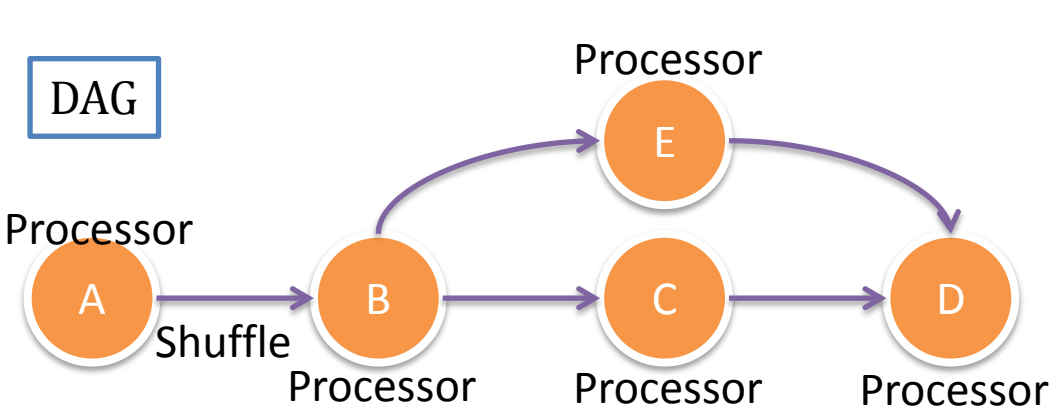
How to Submit an application

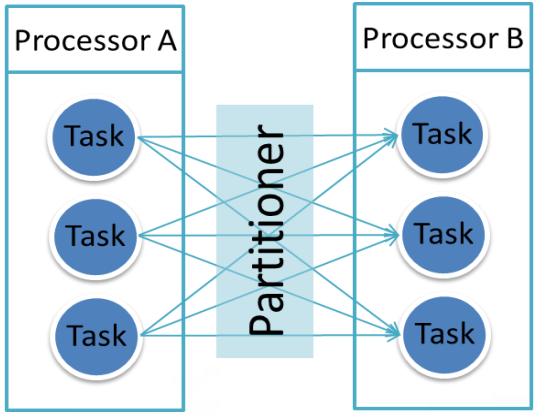


DAG representation and API

DAG API Syntax:

Graph(A~>B~>C~>D, B~>E~>D)





DAG API Example - WordCount

```
val context = new ClientContext()
val split = Processor[Split](splitParallism)
val sum = Processor[Sum](sumParallism)
val app = StreamApplication("wordCount", Graph(split ~> sum), UserConfig.empty)
val appId = context.submit(app)
context.close()
```

```
class Split(taskContext : TaskContext, conf: UserConfig) extends Task(taskContext, conf) {
   override def onNext(msg : Message) : Unit = { /* split the line */ }
}
```

```
class Sum (taskContext : TaskContext, conf: UserConfig) extends Task(taskContext, conf) {
   override def onNext(msg : Message) : Unit = {/* do aggregation on word*/}
}
```

WordCount with **DSL API**

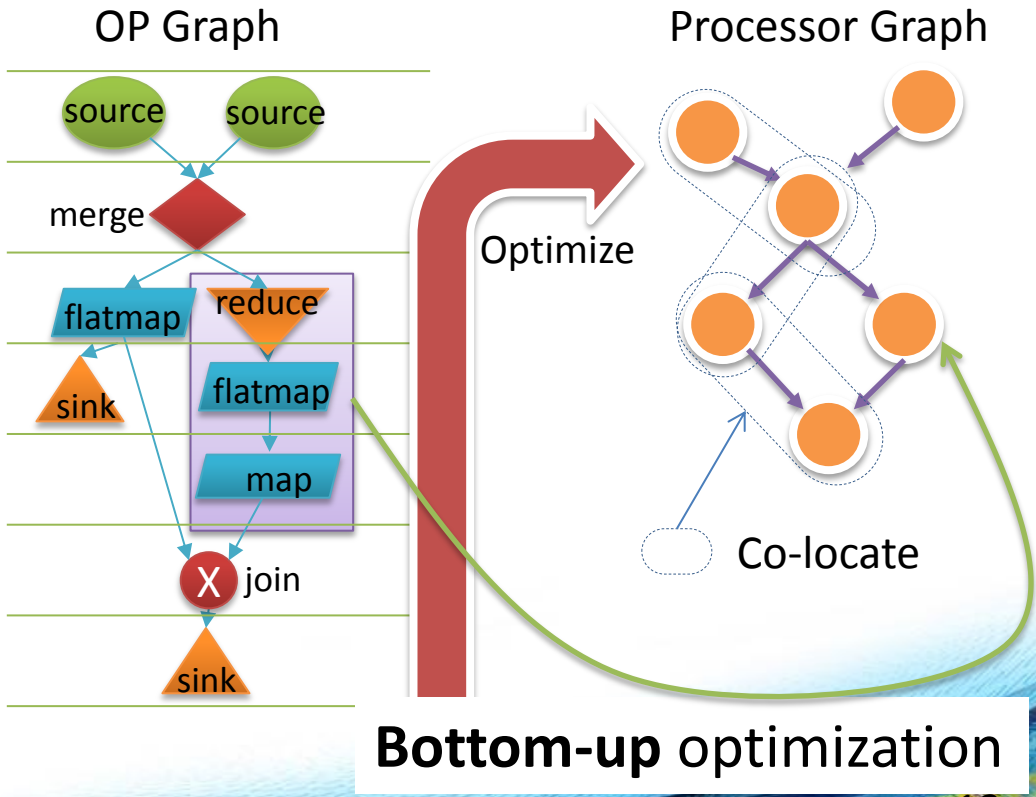
```
val context = ClientContext()
val app = new StreamApp("dsl", context)
val data = "This is a good start, bingo!! bingo!!"
app.fromCollection(data.lines)
  // word => (word, count = 1)
  .flatMap(line => line.split("[\\s]+")).map((_, 1))
  // (word, count1), (word, count2) => (word, count1 + count2)
  .groupByKey().sum.log

val appId = context.submit(app)
context.close()
```

High level DSL API Details

Concepts	Description
StreamApp	OP(Operator) Graph
Stream	path of OP Graph
OP Transformer	Transformation on Stream

Transformer Operators	
flatMap	merge
map	groupBy
filter	process
reduce	



DAG Visualization

DAG Page

Home / Applications / Application2 (dag)

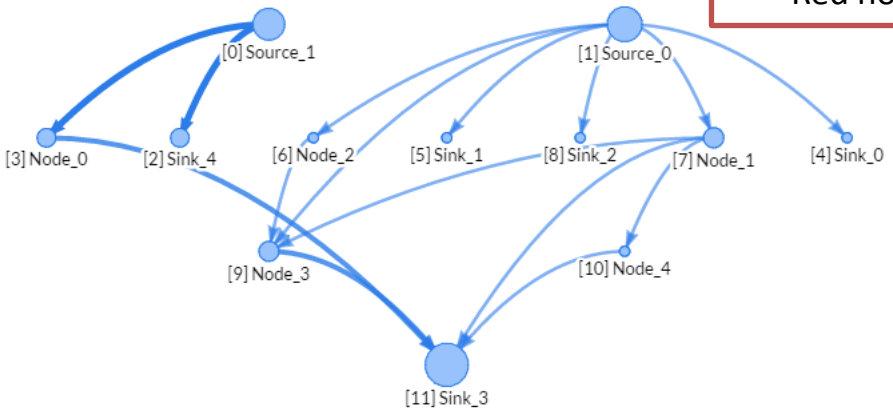
STATUS DAG PROCESSOR METRICS

APPLICATION CLOCK
23:19:24 2015/04/11

Track global min-Clock of all message

DAG:
- Node size reflect throughput
- Edge width represents flow rate
- Red node means something goes wrong

GRAPH: DAG



116.23M messages

SINK PROCESSORS RECEIVE THROUGHPUT
262,362 message/s

SOURCE PROCESSORS SEND THROUGHPUT
224,689 message/s

AVG. PROCESSING TIME PER TASK
0.00 ms

AVG. RECEIVE LATENCY PER TASK
8.30 ms

DAG Visualization

Processor Page

Home / Applications / Application1 (wordCount)

STATUS DAG PROCESSOR METRICS

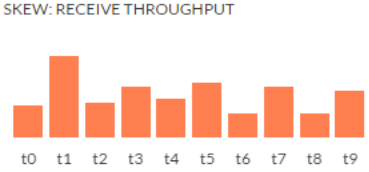
Skew analysis

Task throughput and latency

PROCESSOR #1

Sum

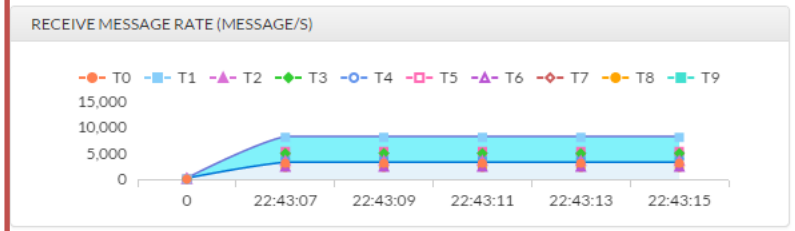
Parallelism 10
Inputs 1
Outputs 0

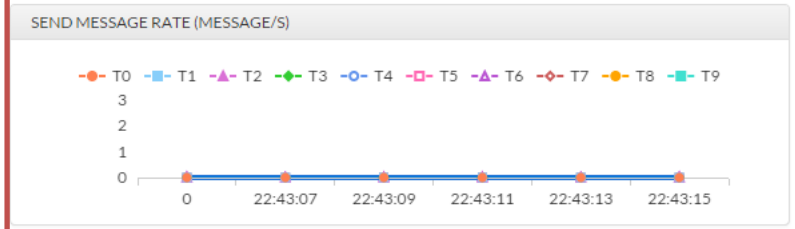


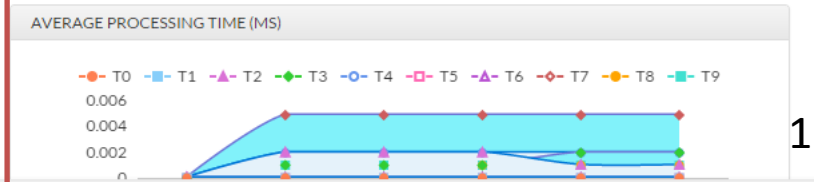
EXECUTORS

ID	ActorPath
0	akka.tcp://app1system0@127.0.0.1:52933/remote/akka.tcp/app1-executor-1@127.0.0.1:52903/user/daemon/appdaemon1/$c/appmaster/executors/0
1	akka.tcp://app1system1@127.0.0.1:52923/remote/akka.tcp/app1-executor-1@127.0.0.1:52903/user/daemon/appdaemon1/$c/appmaster/executors/1

Executor JVM deployment







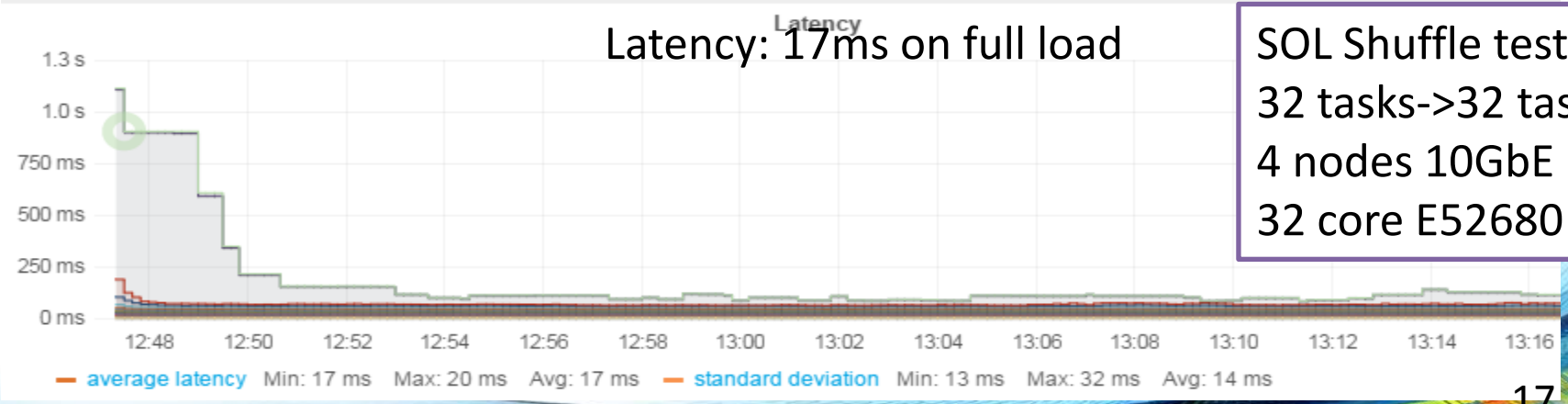
PERFORMANCE TEST

THROUGHOUT、SCALABILITY、FAULT TOLLERANCE

Throughput and Latency

Throughput: 11 million message/second

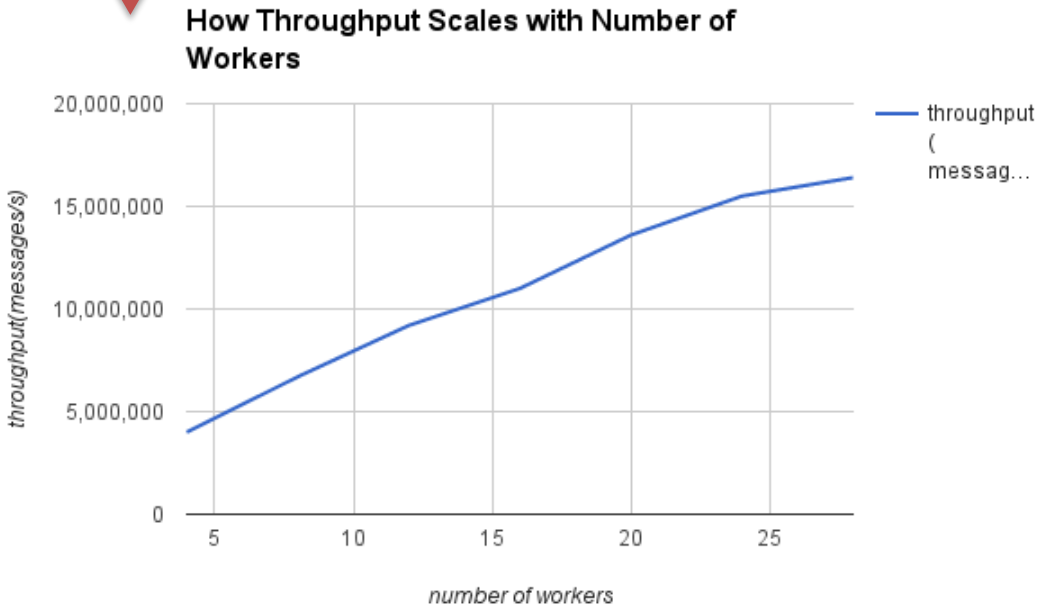
Latency: 17ms on full load

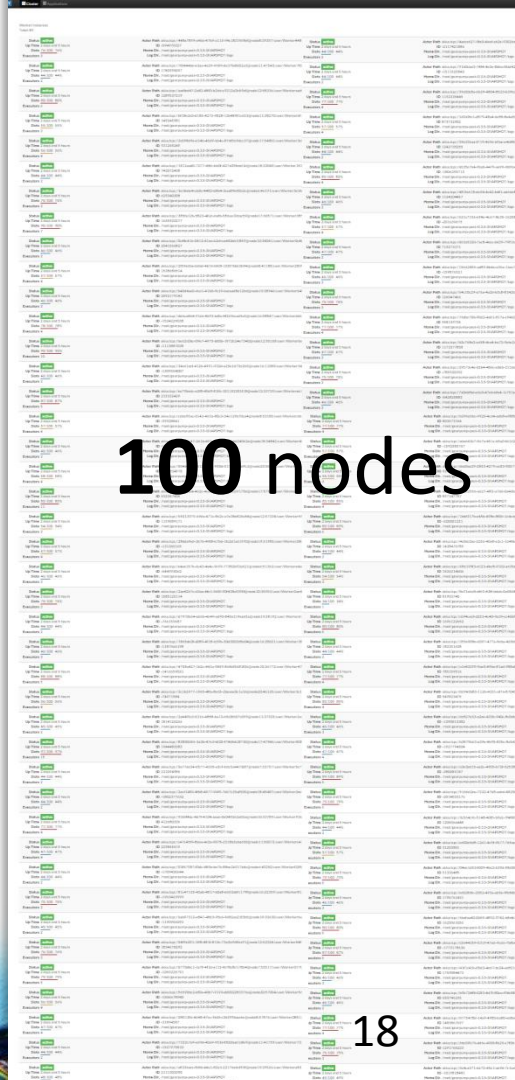


SOL Shuffle test
32 tasks->32 tasks
4 nodes 10GbE
32 core E52680

Scalability

- Test run on **100** nodes and 3000 tasks
- Gearpump performance scales:





100 nodes

Fault-Tolerance: Recovery time

91 worker nodes, 1000 tasks

Failure scenarios	**Recovery time [*]**	**comment**
Cluster Master node Down	0 s	Master HA take effect
Cluster Worker node down	~ 10 seconds	timeout detection take a long time
Message loss	~ 300 ms	Still optimizing Target will be less than 10ms
Application AppMaster down	~ 10 seconds	timeout detection take a log time

Test environment: 91 worker nodes, 1000 tasks (We use 7 machines to simulate 91 worker nodes)

[*]: Recovery time is the time interval between: a) failure happen b) all tasks in topology resume processing data.

GEARPUMP INTERNAL

Overview and general ideas

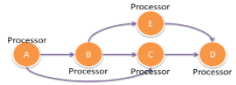
Minclock service

MinClock service track the **min timestamp of all pending** messages in the system **now and future**

Replayable Source

Message(timestamp)

DAG runtime



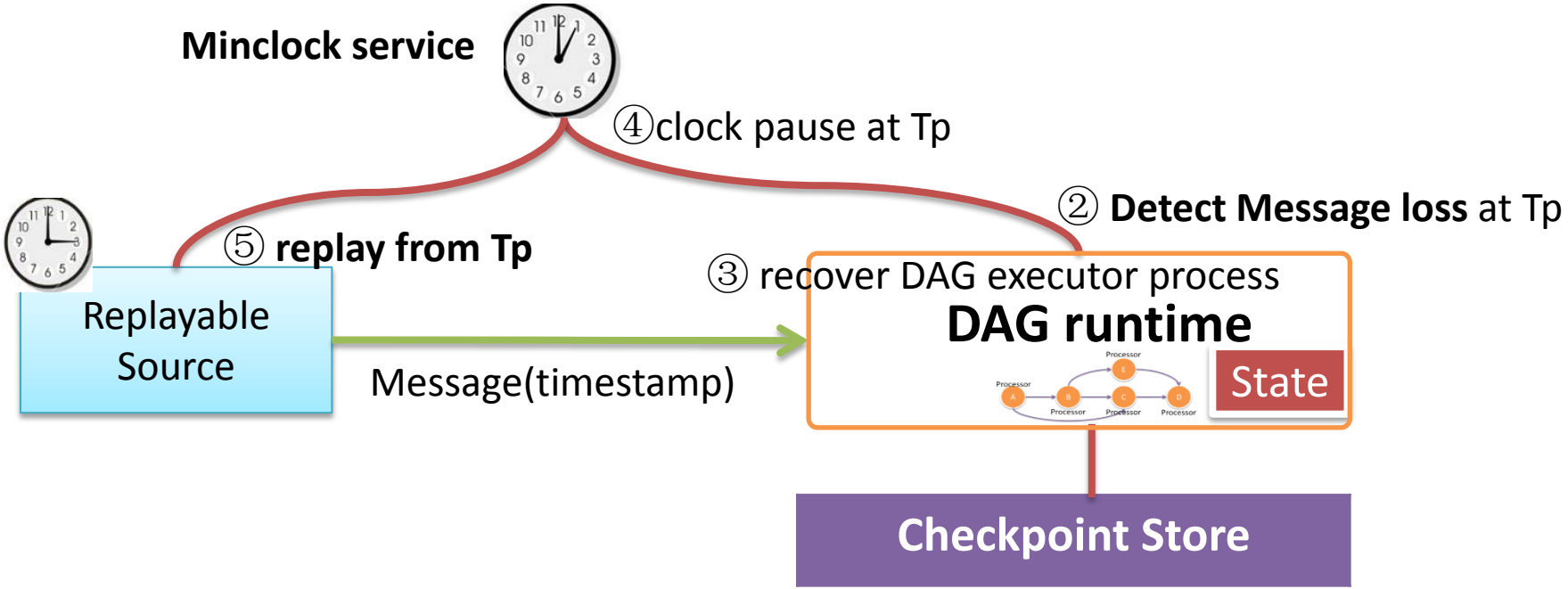
State

① **High Performance Streaming**

Every message in the system Has a **application timestamp**(message **birth time**)

Normal Flow

Overview and general ideas



Recovery Flow

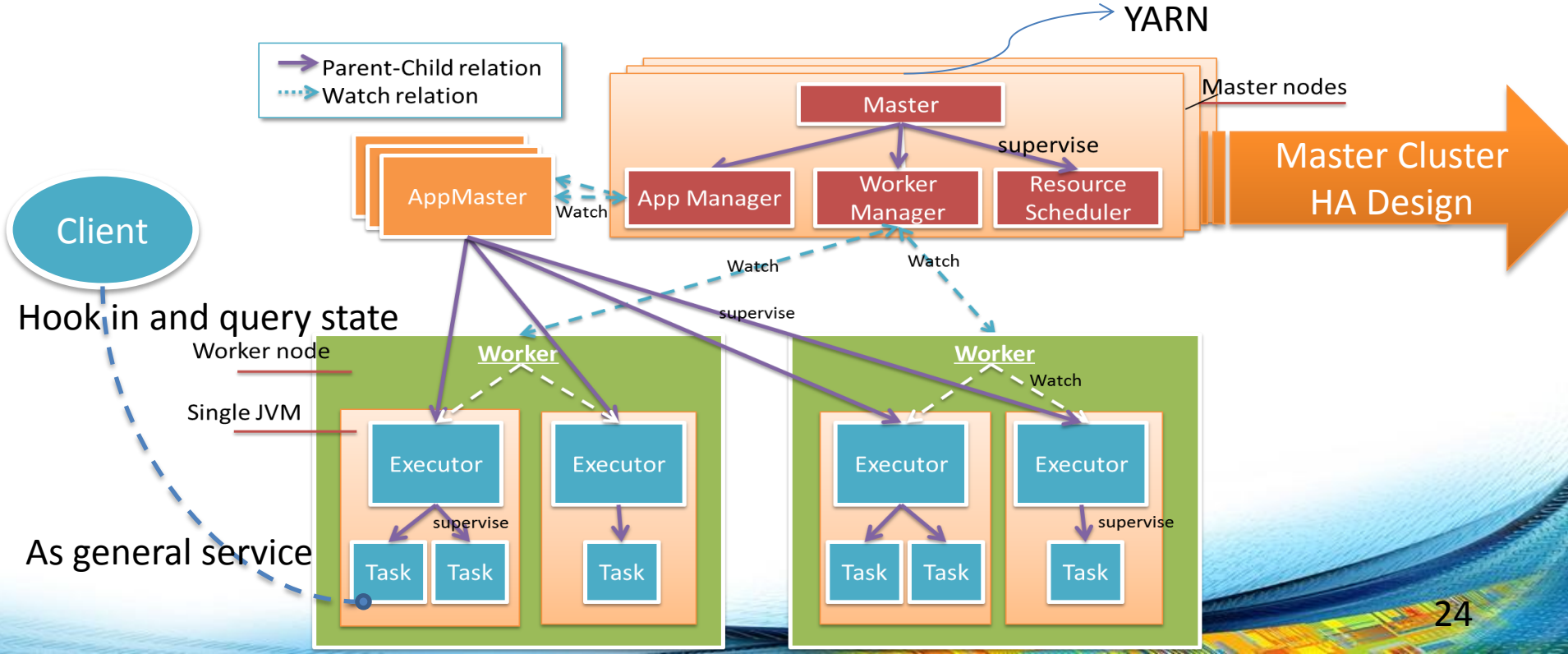
⑥**Exactly-once** State can be reconstructed by message replay:

State(t) = State(t -δ) + Replay all message between(t- δ, t)

1. **High performance streaming**
2. Detect Message loss and other failures
3. DAG Executor Recovery
4. Clock Service, know when message is lost
5. Message replay from clock
6. Exactly-once, de-duplication

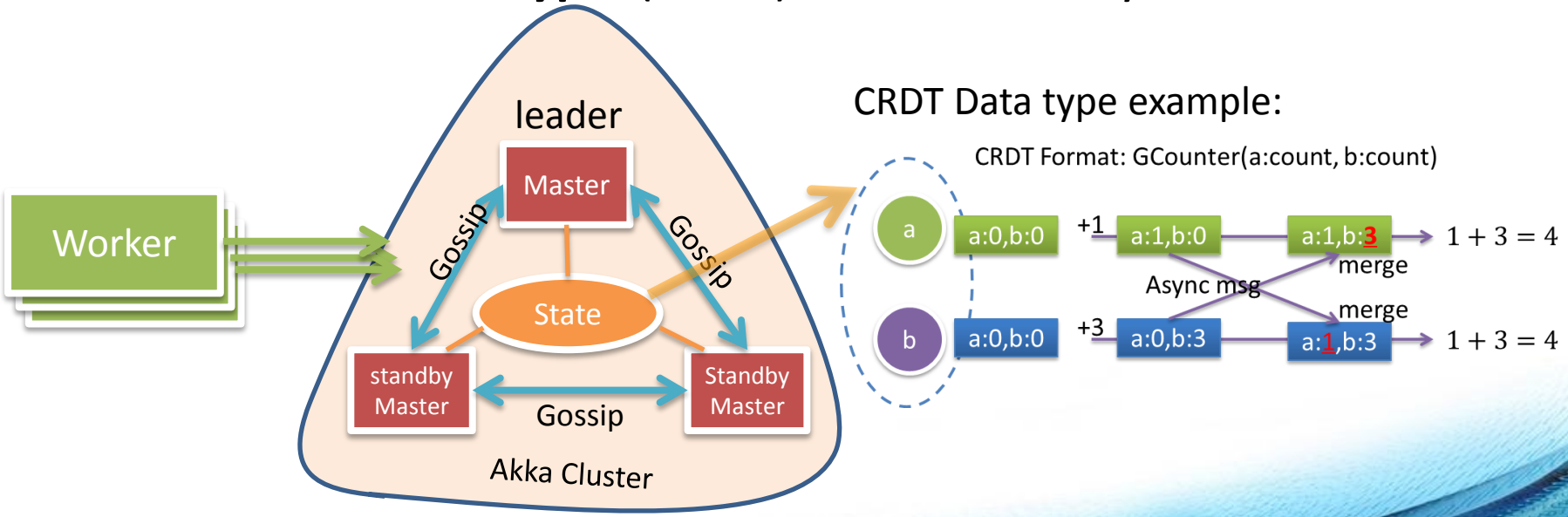
Actor Hierarchy

100% Actor: communication, concurrency, isolation, error handling



Master HA

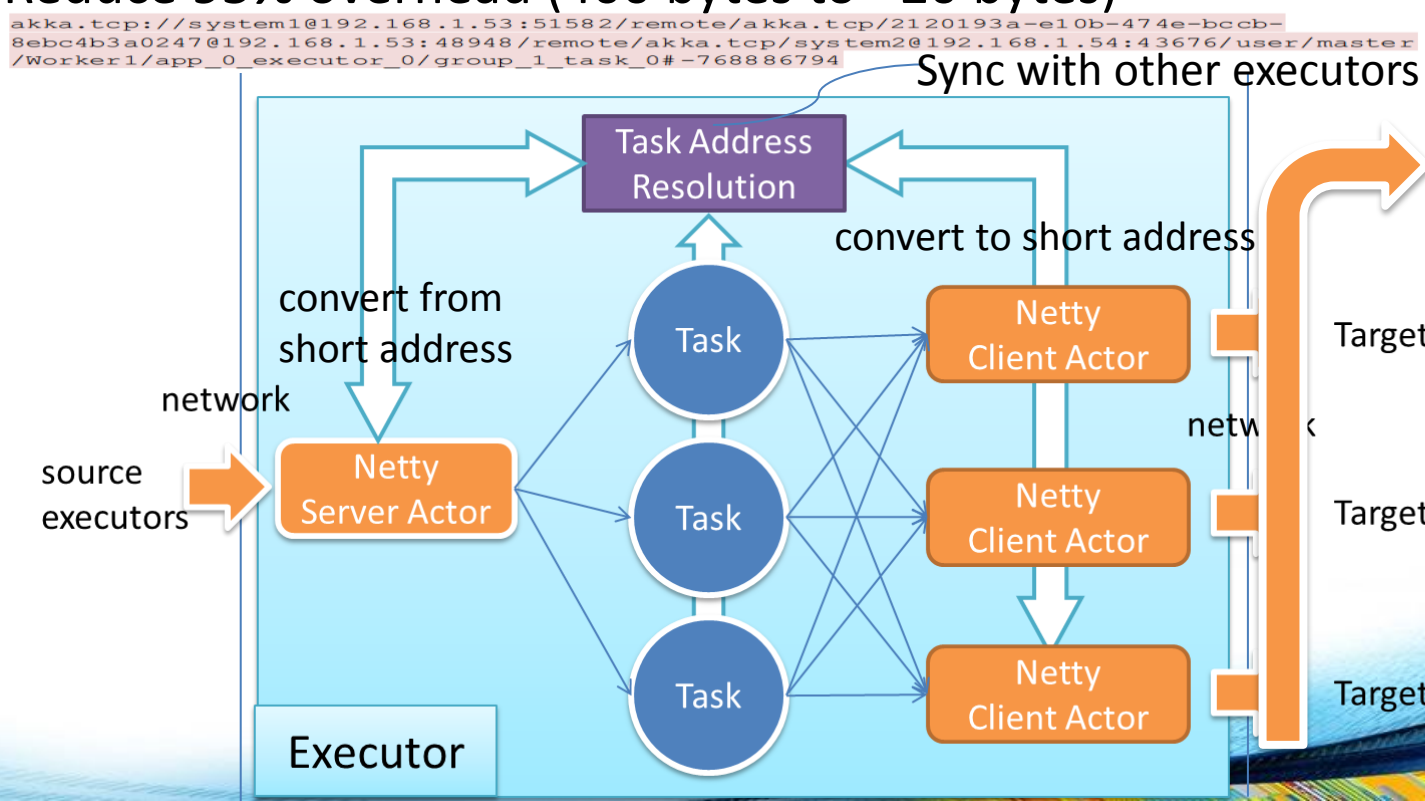
- **Quorum (Majority)**
- **Conflict free data types**(CRDT) for consistency



Decentralized: No central meta server

High performance Messaging Layer

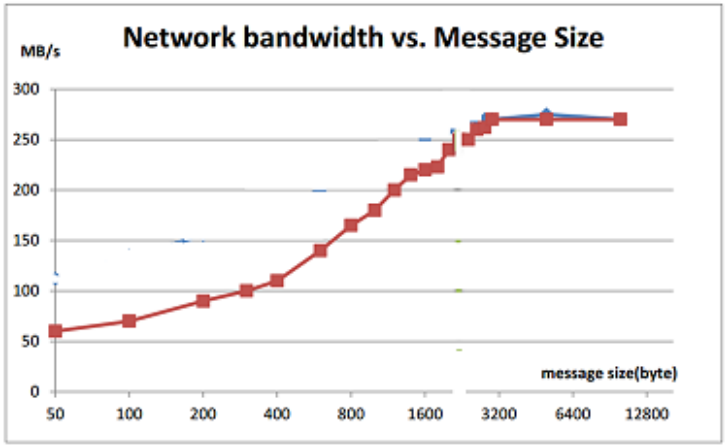
- Akka remote message has a big overhead, (sender + receiver address)
- Reduce **95%** overhead (400 bytes to ~20 bytes)



Effective batching

Network **Idle**: Flush **as fast as we can**

Network **Busy**: **Smart batching** until the network is open again.



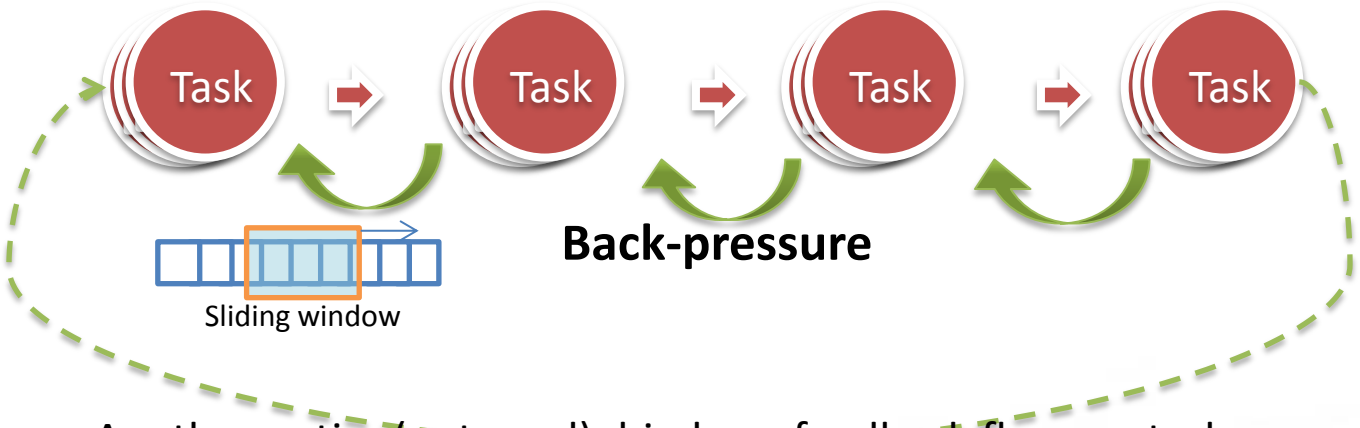
Network Bandwidth **Doubled**

For 100 byte per message

This feature is ported from Storm-297

Flow Control

Pass back-pressure **level-by-level**



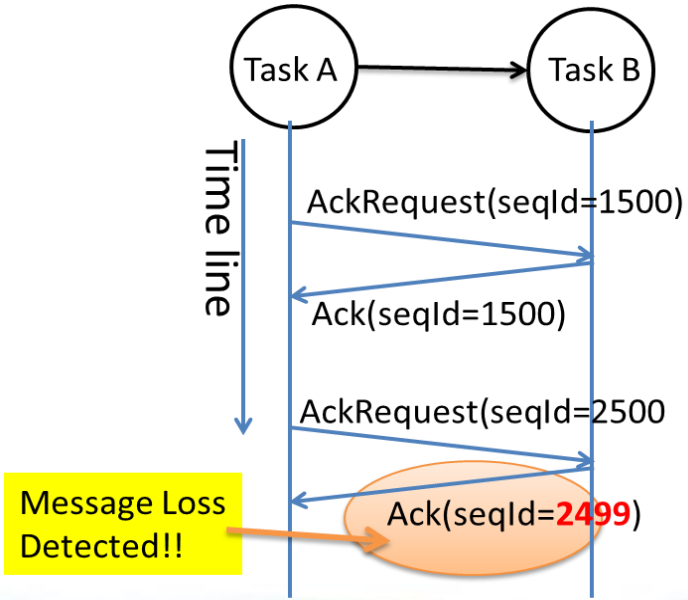
Another option(not used): big-loop-feedback flow control

1. High performance streaming
2. **Detect Message loss and other failures**
3. DAG Recovery
4. Clock Service, know when message is lost
5. Message replay from clock
6. Exactly-once, de-duplication

Failure Detection

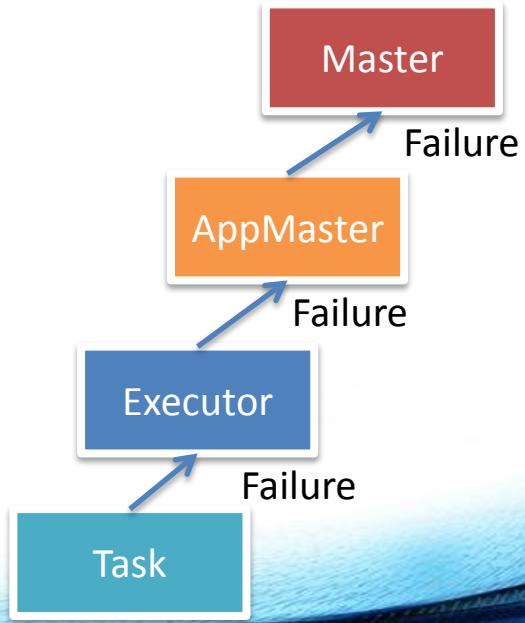
For Message loss:

- **AckRequest and Ack**



For JVM Crash, Network Down:

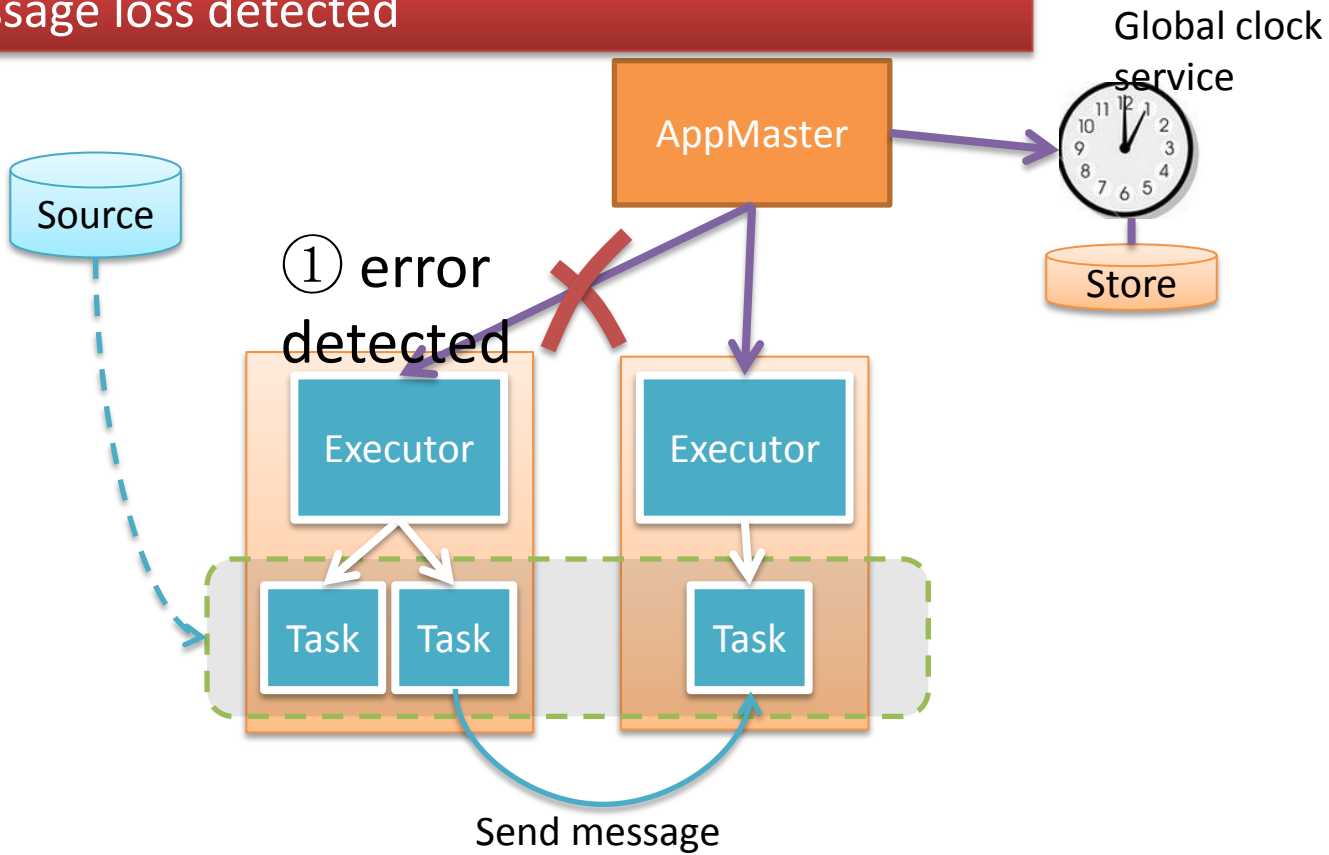
- **Actor Supervision**



1. High performance streaming
2. Detect Message loss and other failures
3. **DAG Recovery**
4. Clock Service, know when message is lost
5. Message replay from clock
6. Exactly-once, de-duplication

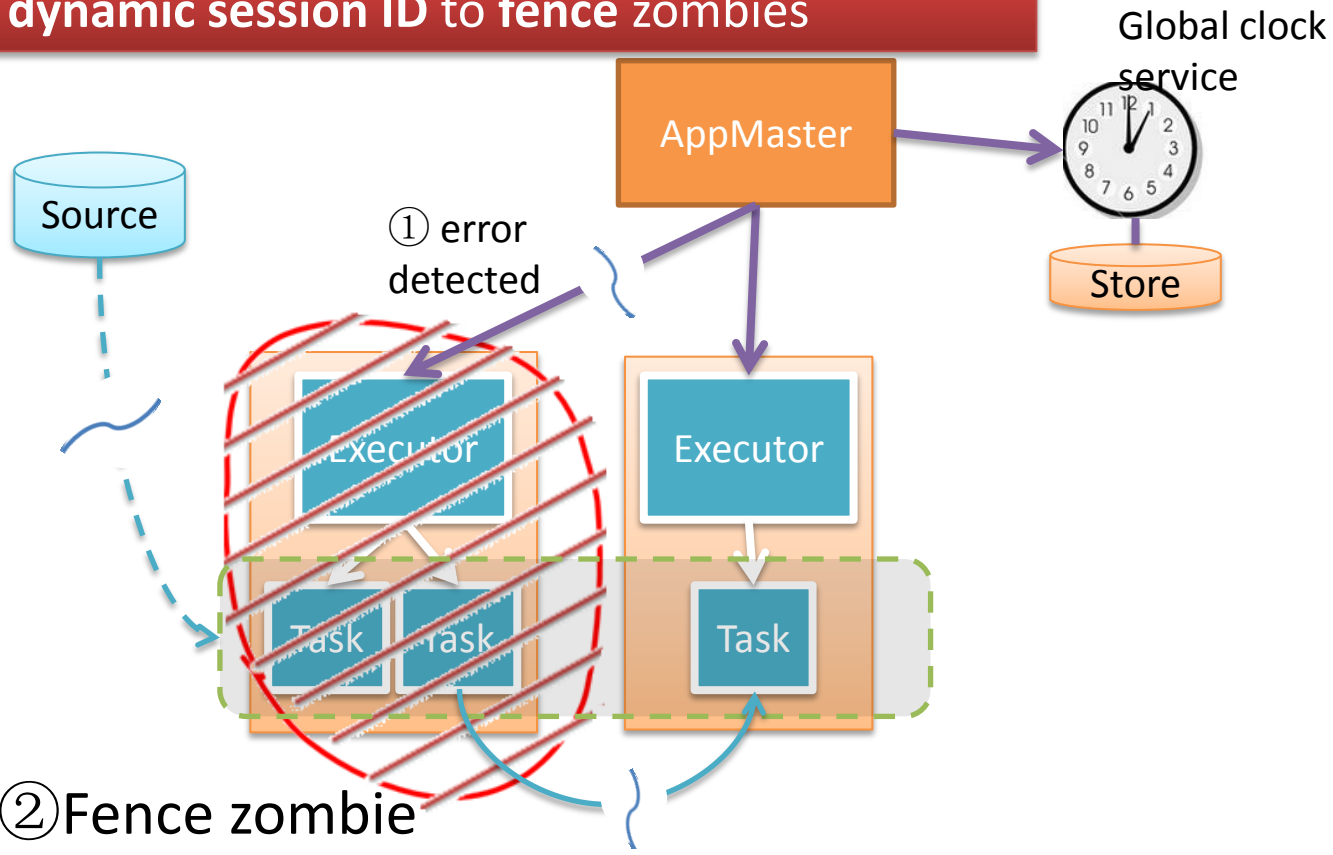
DAG Recovery: Quarantine and Recover

1. Message loss detected



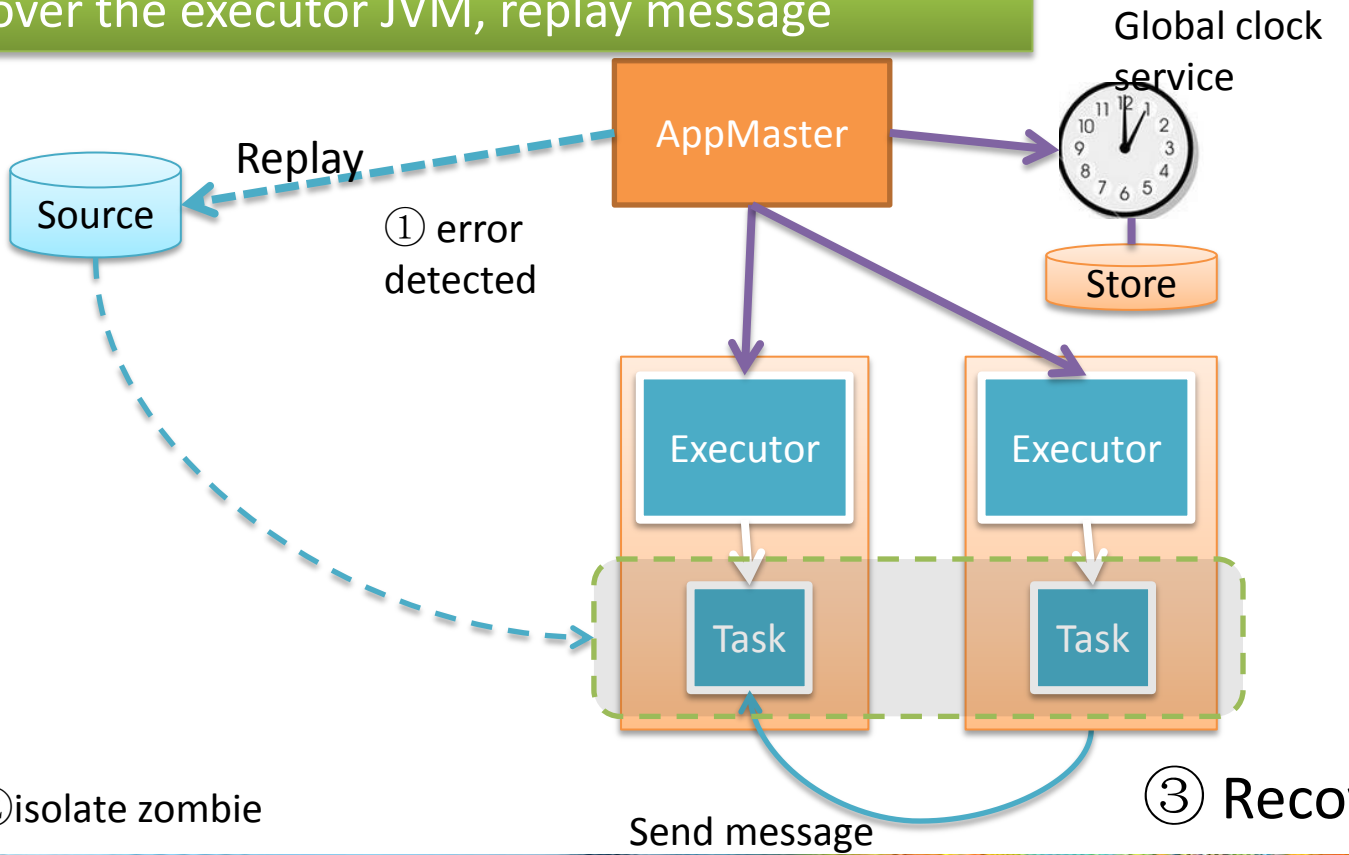
DAG Recovery: Quarantine and Recover

2. Use **dynamic session ID** to **fence** zombies



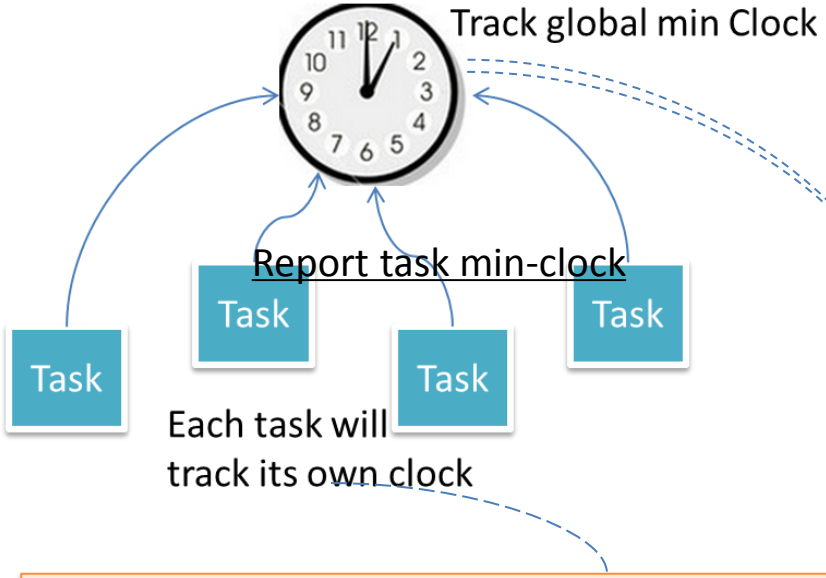
DAG Recovery: Quarantine and Recover

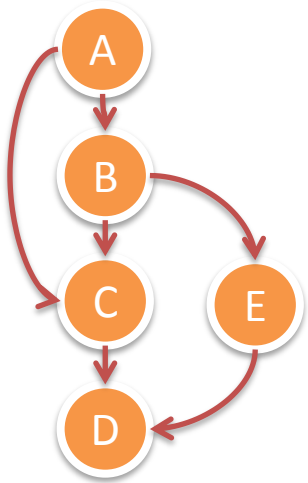
3. Recover the executor JVM, replay message



1. High performance streaming
2. Detect Message loss and other failures
3. DAG Recovery
4. **Clock Service, know when message is lost**
5. Message replay from clock
6. Exactly-once, de-duplication

Global Clock Service – track application min-Clock (1)

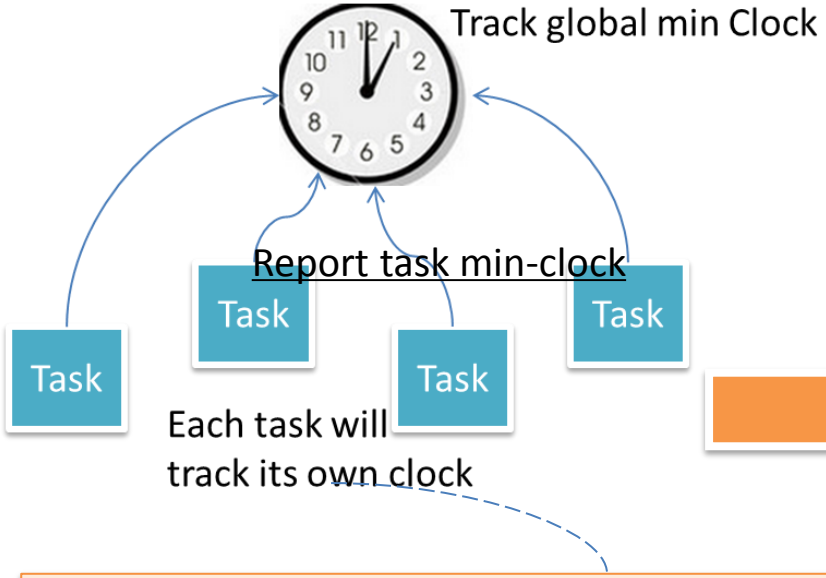




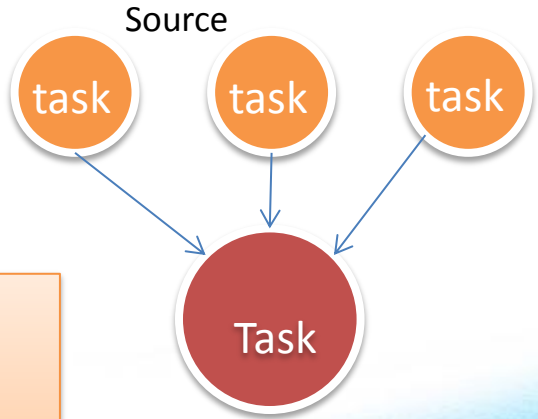
Definition*:* **Task min-clock** is
Minimum of (
min timestamp of pending-messages in current task
Task min-Clock *of all upstream tasks*
)

Min-Clock of D is min-clock of global

Global Clock Service – track application min-Clock (2)



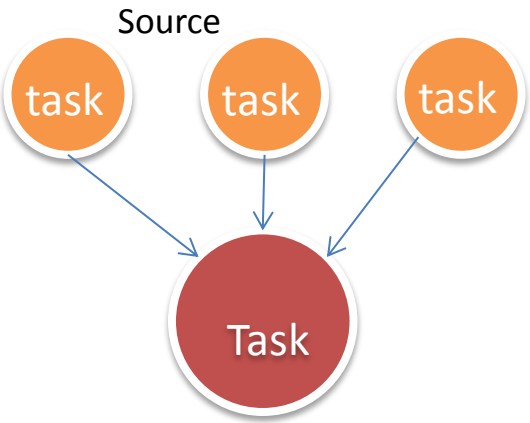
One Task can have thousands upstream tasks, how to effectively track all of them?



Definition: **Task min-clock** is
Minimum of (
min timestamp of pending-messages in current task
***Task min-Clock** of all upstream tasks*
)

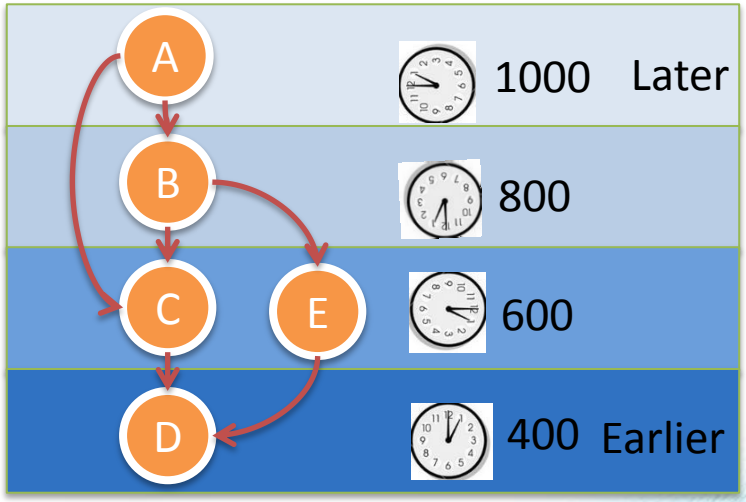
Global Clock Service – track application min-Clock (3)

One Task can have thousands upstream tasks, how to effectively track all of them?



Implementation

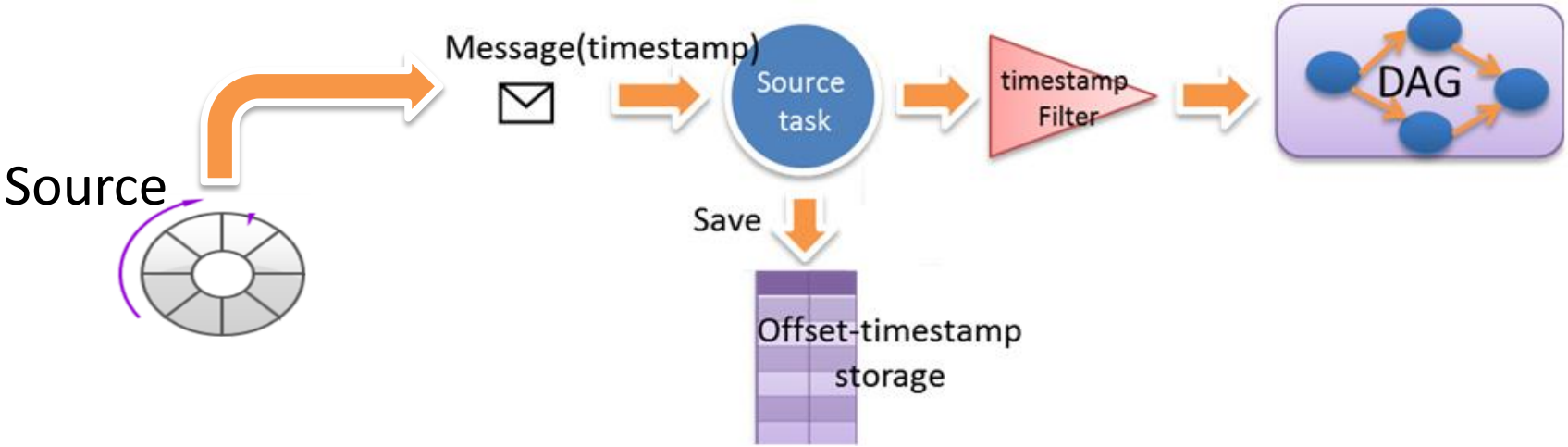
Level Clock Ever incremental



1. High performance streaming
2. Detect Message loss and other failures
3. DAG Executor Recovery
4. Clock Service, know when message is lost
5. **Message replay from clock**
6. Exactly-once, de-duplication

Source-based Message Replay

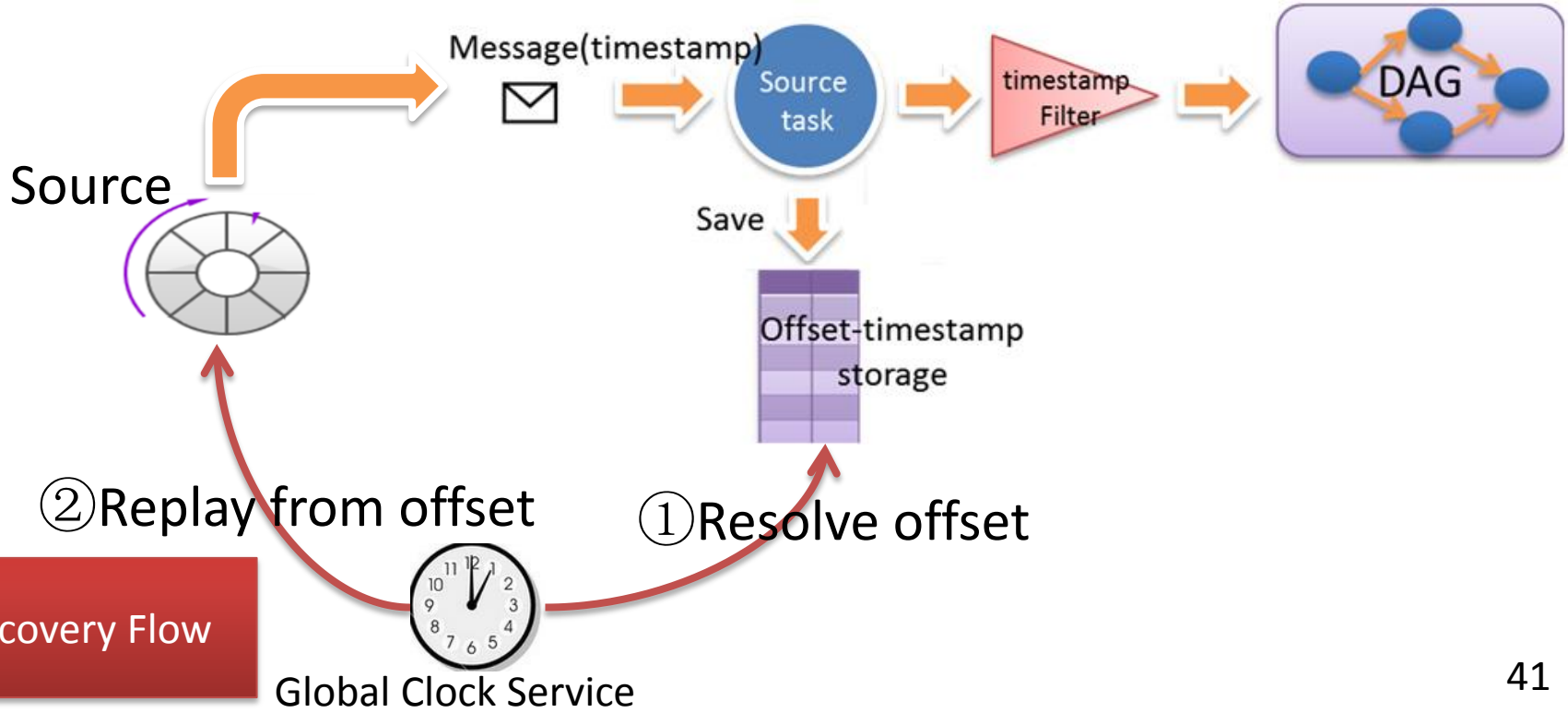
Replay from the **very-beginning** source



Normal Flow

Source-based Message Replay

Replay from the **very-beginning** source



1. High performance streaming
2. Detect Message loss and other failures
3. DAG Executor Recovery
4. Clock Service, know when message is lost
5. Message replay from clock
6. **Exactly-once, de-duplication**

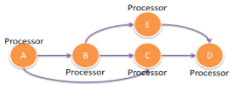
Exactly-once message processing

State(t) = State(t -δ) + Replay all message between(t- δ, t)

Key: Ensure State(t) only contains message(timestamp <= t)

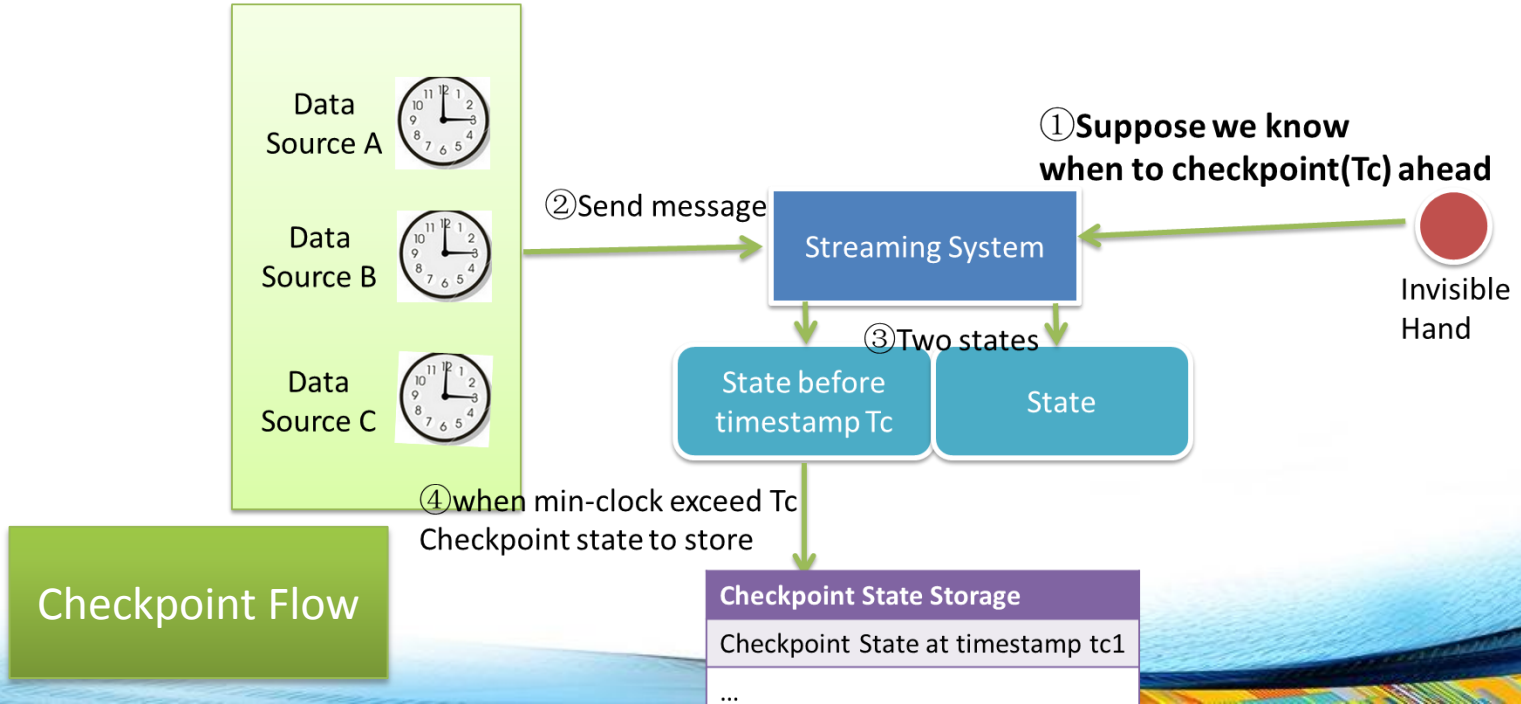
How?

DAG runtime

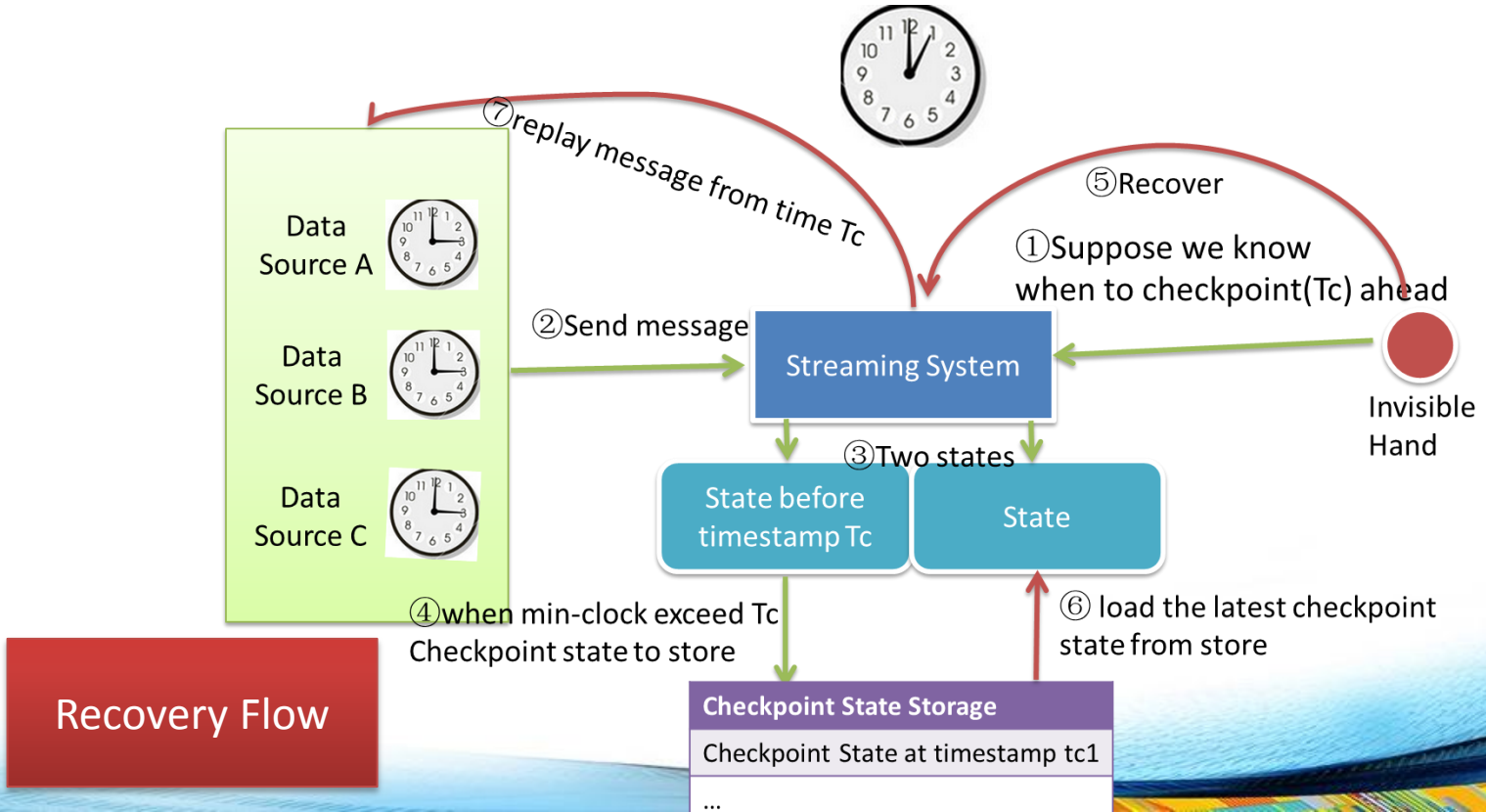


Checkpoint Store

Exactly-once message processing



Exactly-once message processing

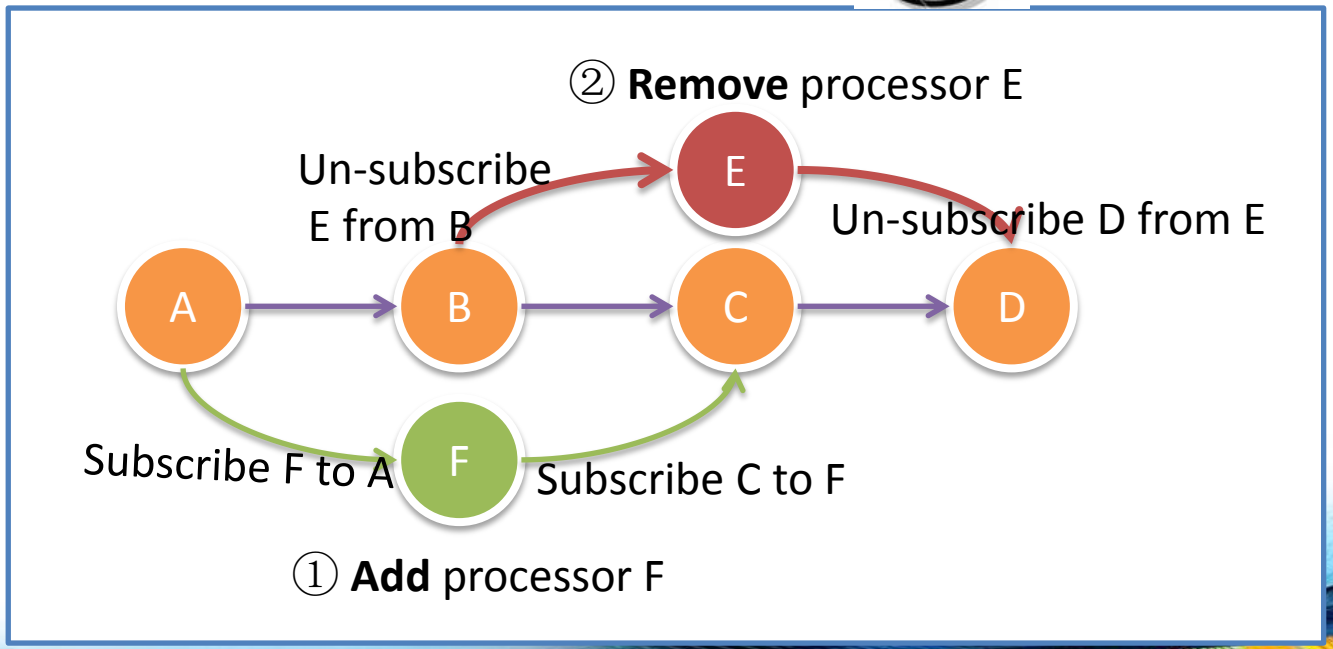


Recovery Flow

Dynamic DAG

Use Pub-Sub model

Maintain the correct min clock during transition

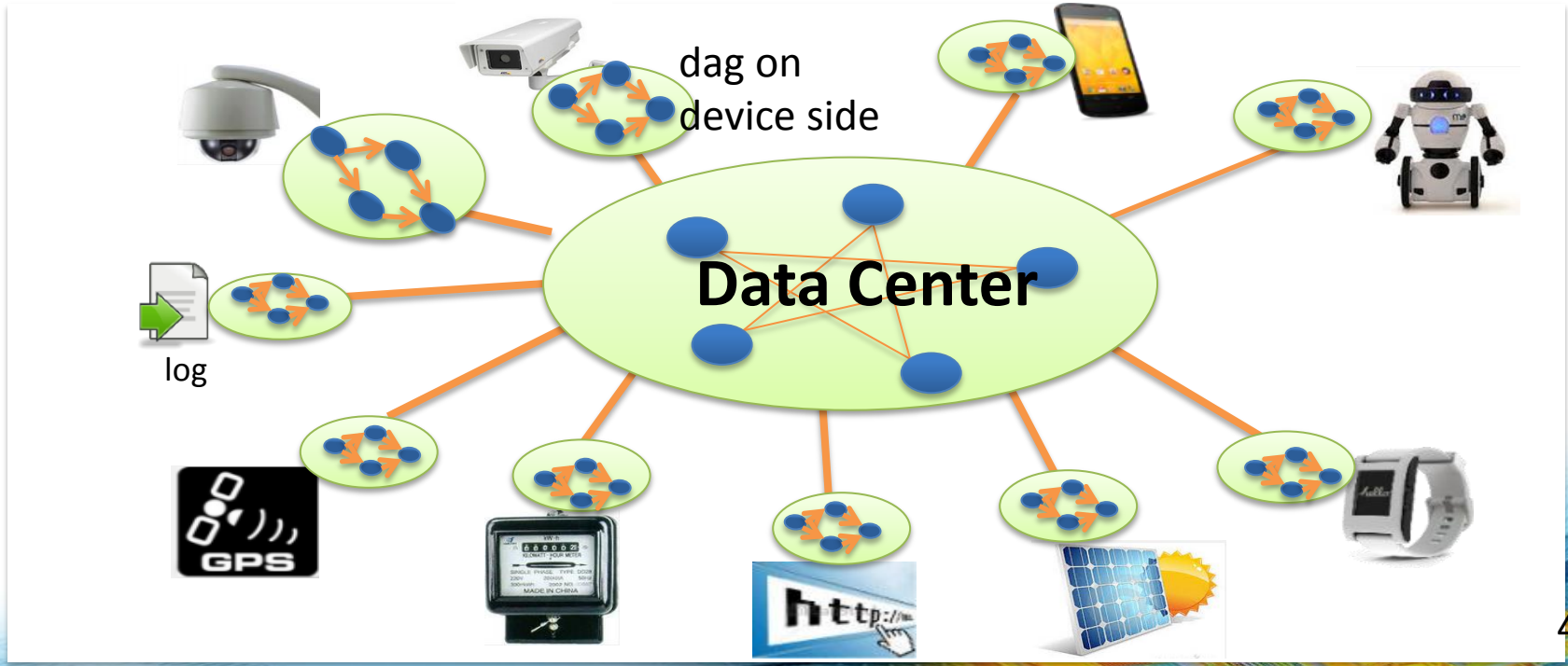


UNIQUE USE CASES
独特用例

IOT Transparent Cloud

Target Problem	Large gap between edge device with data center

Location transparent. Same programming model on IOT device



Unified log ingestion and processing

Target Problem Distributed application is broken into many isolated pieces

Single Gearpump Application For All

Deploy agents

Manage processing

Manage query service

Agent

Agent

Agent

Kafka queue

DAG

E

B

C

D

Processor

Processor

Processor

Query

Visualization

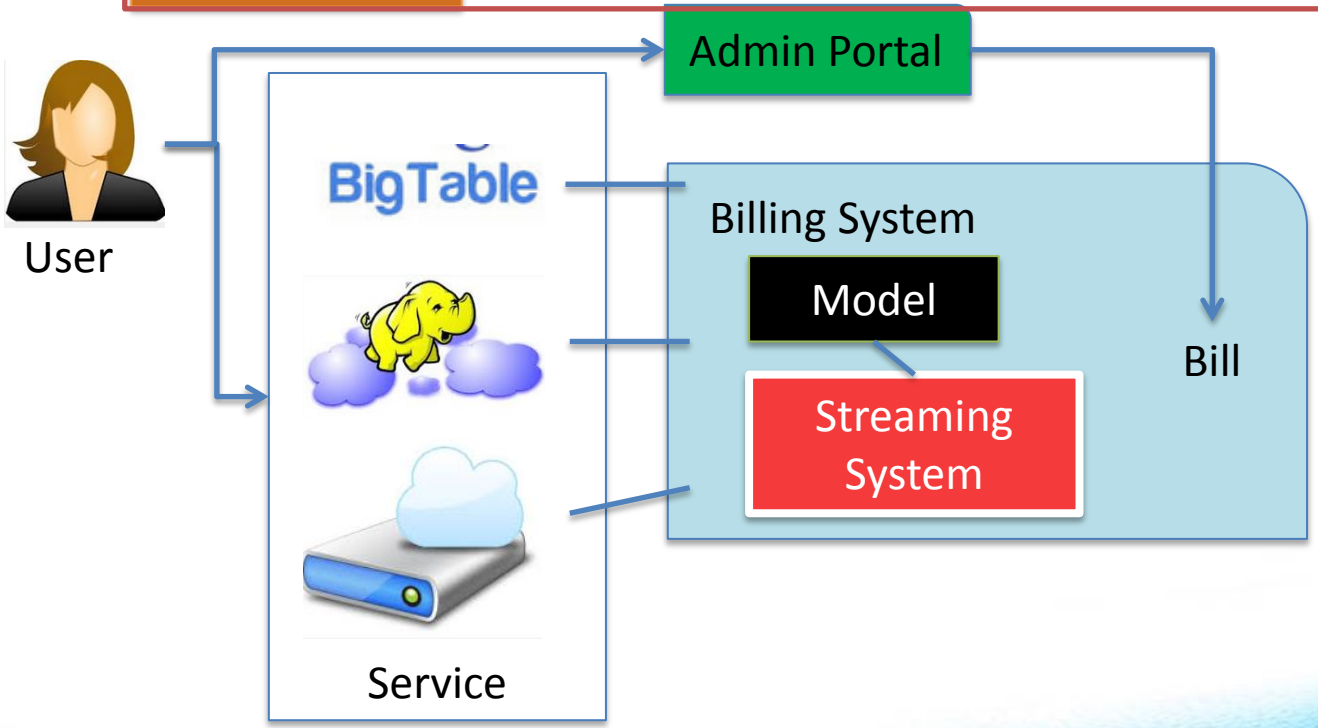
Collect logs from different places

HDFS

APACHE HBASE

Exactly-once: Financial use cases

Target Problem	**transactional** exactly-once real-time streaming system

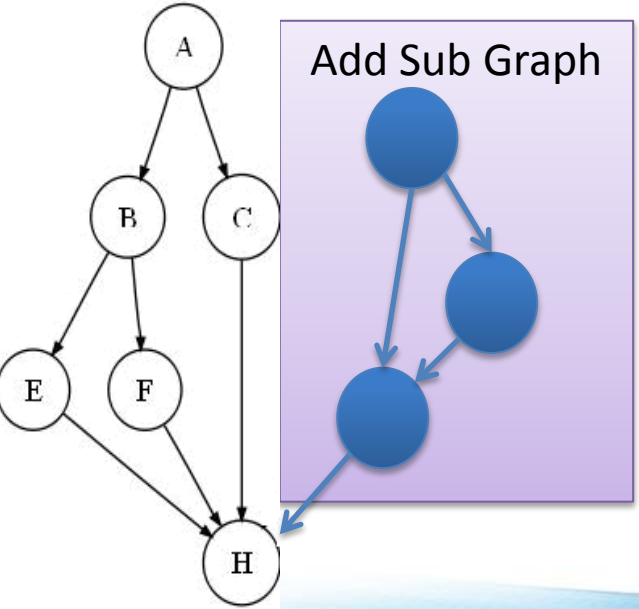


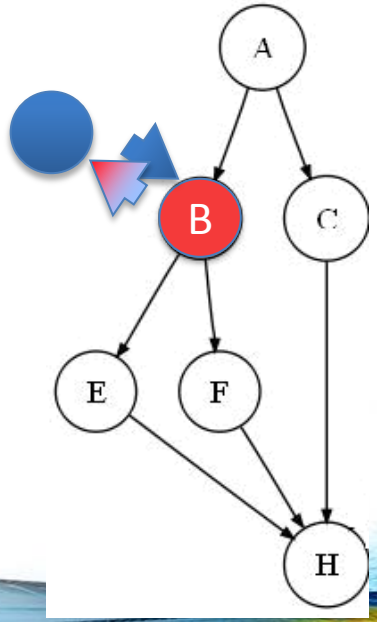
Transformers: **Dynamical** DAG

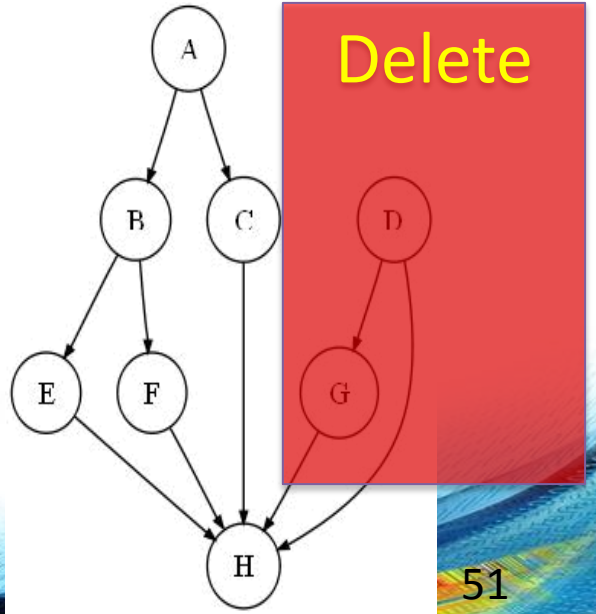
Target Problem No existing way to manipulate the DAG on the fly

Manipulate the DAG **on the fly**

- Dynamic Attach
- Dynamic Replace
- Dynamic Remove



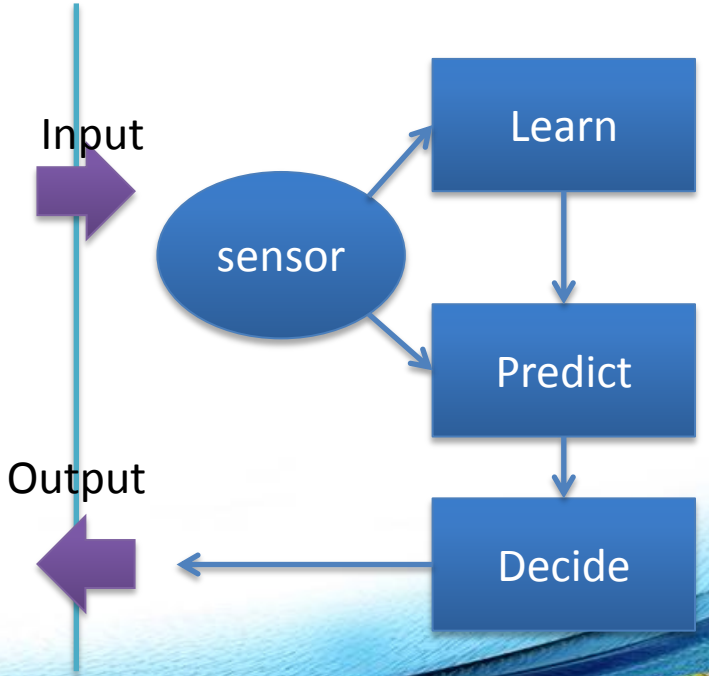




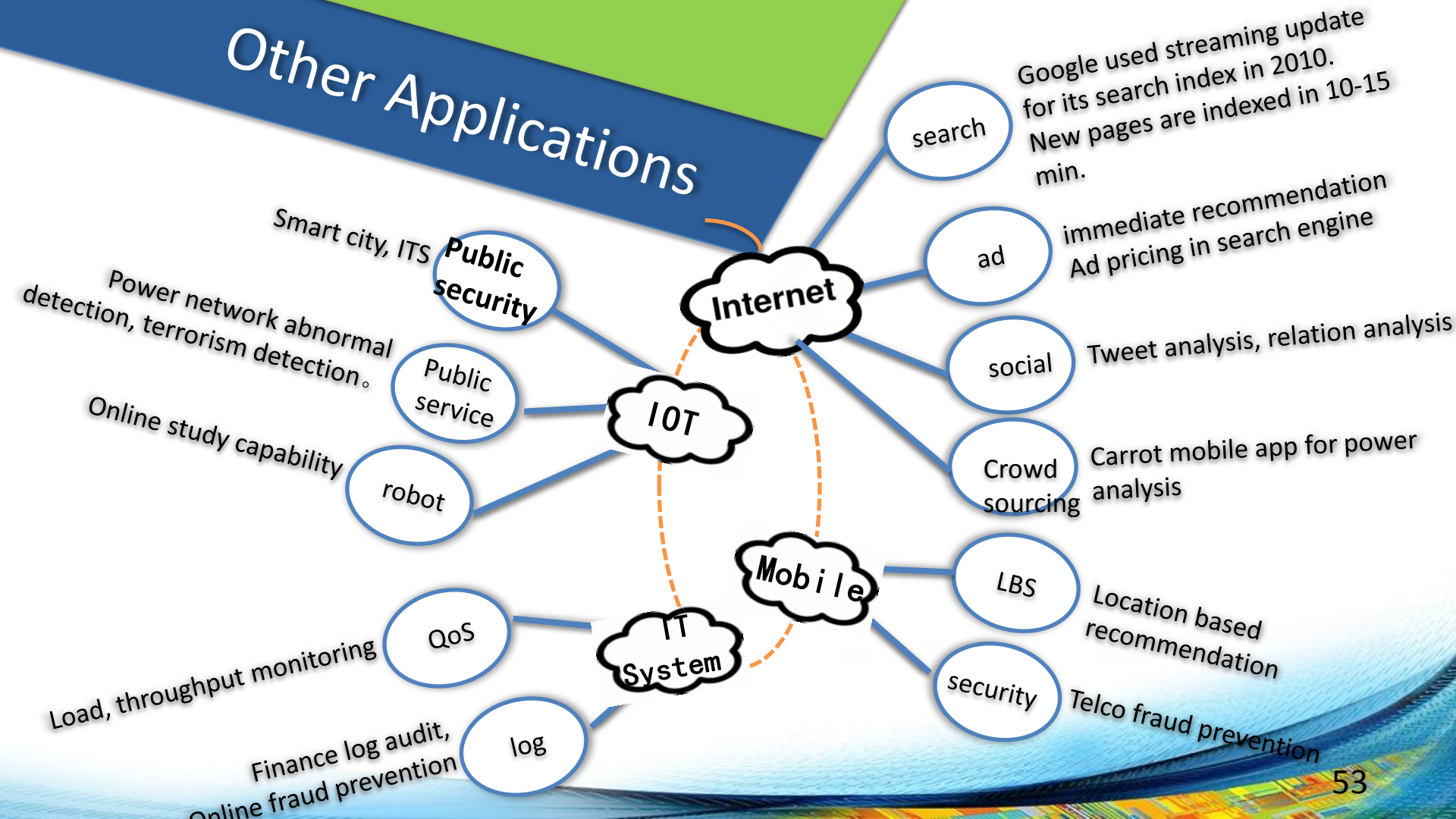
Eve: **Online** Machine Learning

Target Problem ML train and predict online in real-time for decision support.

- **Decide immediately** based **online learning** result



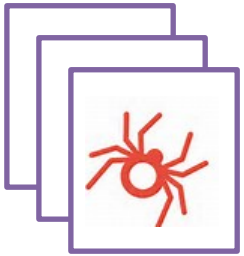
Other Applications



APPLICATION AND DEMO

Example1: Stock Drawdown Tracking

More than 3000 index



Crawlers

Group by Stock id

Publish Alerts

Query

Drawdown Definition

Example2: Intelligent Traffic System

3000 cross roads

专线

Gateway

HDFS

KV Query

APACHE HBASE

Realtime analysis

DAG

- Travel time
- Fake Plate
- Over speed
- Traffic planning
- Real time statistics

Other demos

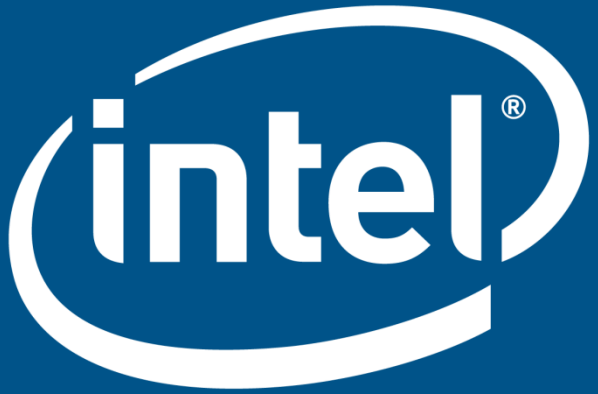
- complex Graph
- Stock data analysis (Drawdown tracking)
- ITS
- Scalability, 100 nodes, 1,000,000 tasks
- DSL

Status & Plan

- Our goal: Make this an Apache project
  - Welcome code contribution!
  - http://gearpump.io
- Plan:
  - **Connect**: IOT
  - **Platform**: Dynamic DAG, Exactly-once, etc. (release soon)
  - **Data:** Real-time analysis algorithm

References

- 钟翔 大数据时代的软件架构范式：Reactive架构及Akka实践, 程序员期刊2015年2A期
- Gearpump whitepaper http://typesafe.com/blog/gearpump-real-time-streaming-engine-using-akka
- 吴甘沙 低延迟流处理系统的逆袭, 程序员期刊2013年10期
- Stonebraker http://cs.brown.edu/~ugur/8rulesSigRec.pdf
- https://github.com/intel-hadoop/gearpump
- Gearpump: https://github.com/intel-hadoop/gearpump
- http://highlyscalable.wordpress.com/2013/08/20/in-stream-big-data-processing/
- https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines
- Sqlstream http://www.sqlstream.com/customers/
- http://www.statsblogs.com/2014/05/19/a-general-introduction-to-stream-processing/
- http://www.statalgo.com/2014/05/28/stream-processing-with-messaging-systems/
- Gartner report on IOT http://www.zdnet.com/article/internet-of-things-devices-will-dwarf-number-of-pcs-tablets-and-smartphones/


intel