GearPump – Real-time Streaming Engine Using **Akka***

Sean Zhong, Kam Kasravi, Huafeng Wang, Manu Zhang, Weihua Jiang

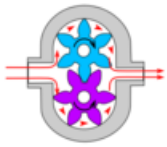
{xiang.zhong, kam.d.kasravi, huafeng.wang, tianlun.zhang, weihua.jiang}@intel.com

Intel Big Data Technology Team

Dec. 2014

About GearPump

GearPump (https://github.com/intel-hadoop/gearpump) is a lightweight, real-time, big data streaming engine. It is inspired by recent advances with Akka and a desire to improve on existing streaming frameworks. GearPump draws from a number of existing frameworks including MillWheel[1], Apache Storm[2], Spark Streaming[3], Apache Samza[4], Apache Tez[5], and Hadoop YARN[6] while leveraging Actors throughout its architecture.



The name GearPump is a reference to the engineering term "gear pump," which is a super simple pump that consists of only two gears, but is very powerful at streaming water.

In this article, we will present how we've leveraged Akka to build this engine.

Motive

Today's big data streaming frameworks are in high demand to process immense amounts of data from a rapidly growing set of disparate data sources. Beyond the requirements of fault tolerance, persistence and scalability, streaming engines need to provide a programming model where computations can be easily expressed and deployed in a distributed manner. GearPump meets this objective while differentiating itself from other streaming engines by elevating and promoting the Actor Model as the primary entity permeating the framework. What we set out to build is a simple and powerful streaming framework. The Scala language and Akka offer a higher level of abstraction allowing frameworks to focus more on the application and make the engine more lightweight. We've adhered to several basic principles:

- **Message-driven architecture is the right choice for real-time streaming.** Real-time applications need to respond to messages immediately. Akka gives us all the facilities and building blocks for message-driven architecture; it simplifies the design and coding and allows code to evolve more quickly.

- **Actor is the right level of concurrency**. It can give us better performance to scale up and scale out. We have seen other big data engines using process, thread, or self-managed thread pools to parallelize. Actors are much more lightweight. Moreover, they have many optimizations related to concurrency; Actor execution can be swapped in and out efficiently with respect to fairness and performance.

- **The flexibility of the Actor Model imbues streaming with some unique abilities.** To see an example of this, let's take a look at a video streaming problem in the next section.

Example

Suppose we are using streaming to solve a public security problem: we want to find and track a crime suspect's location by correlating all video data from HD cameras across the city. This requires processing raw video data in order to run face recognition algorithms on this data, but we cannot

afford to pass ALL data back to the data center as the bandwidth is limited. Figure 1 shows how GearPump approaches this problem:

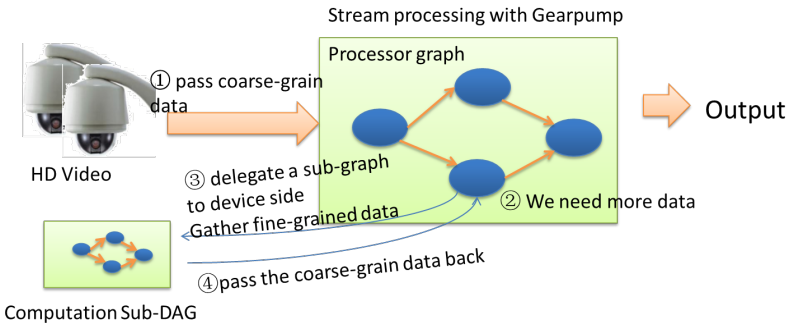


Figure 1: *Video Stream Processing, and Dynamic Graph*

1. Cameras pass coarse-grained feature data to the backend data center. Raw video data cannot be transferred because bandwidth usage is limited.
2. The backend correlates some of the data with nearby cameras and identifies a potential suspect. This requires additional examination of all related video frames for this target.
3. A task is **dynamically** delegated to one or more edge devices to gather the raw video data and send it back. The backend then runs face recognition algorithms on this data and passes results downstream.

The interesting part of the use case is step 3, where new processing directives are dynamically deployed to specific edge devices in order to allow subsequent ingestion and analysis. If we generalize this capability, we can transform a **DAG** at **runtime where relevant computation can be deployed beyond the boundary of the data center**. The location transparency of the Actor Model makes this possible.

Overview

In this section we describe how we model streaming within the Actor Model (Figure 2). Note there is an obvious parallelism to YARN; however, our model is concentric to the application—something YARN architects explicitly delegate to "a host of framework implementations[7]." Integration with YARN is on our roadmap and will pair a compelling application streaming architecture with YARN's broadly recognized de-facto resource management architecture.

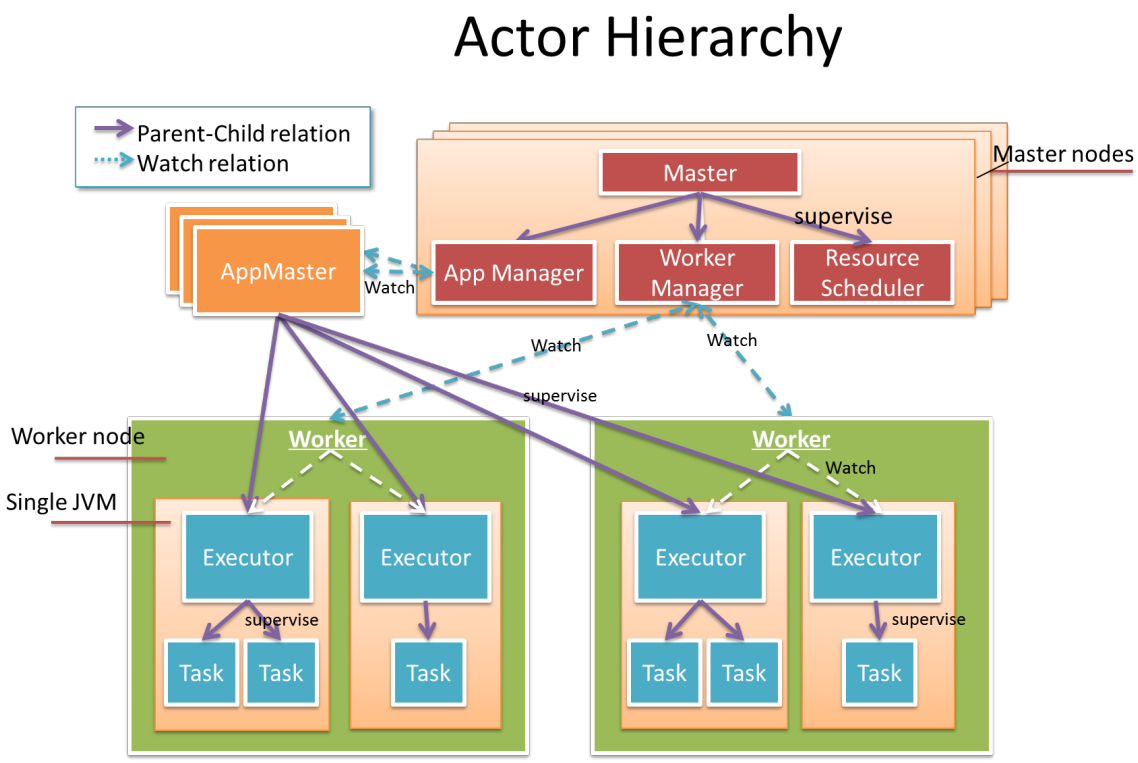


Figure 2: *GearPump Actor Hierarchy*

Everything in the diagram is an actor. The actors fall into two categories, Cluster and Application Actors.

Cluster Actors

- **Worker**: Maps to a physical worker machine. It is responsible for managing resources and reporting metrics on that machine.
- **Master**: Heart of the cluster, which manages workers, resources, and applications. The main function is delegated to three child actors: App Manager, Worker Manager, and Resource Scheduler.

Application Actors

- **AppMaster**: Responsible for scheduling the tasks to workers and managing the state of the application. Different applications have different AppMaster instances and are isolated.
- **Executor**: Child of AppMaster, represents a JVM process. Its job is to manage the lifecycle of tasks and recover them in case of failure.
- **Task**: Child of Executor, does the real job. Every task actor has a global unique address. One task actor can send data to any other task actors. This gives us great flexibility of how the computation DAG is distributed.

All actors in the graph are woven together with actor supervision; actor watching and all errors are handled properly via supervisors. In a master, a risky job is isolated and delegated to child actors,

so it's more robust. In the application, an extra intermediate layer, "Executor," is created so that we can do fine-grained and fast recovery in case of task failure. A Master watches the lifecycle of AppMaster and Worker to handle the failures, but the lifecycle of Worker and AppMaster are not bound to a Master Actor by supervision, so that Master node can fail independently. Several Master Actors form an Akka cluster, the Master state is exchanged using the Gossip protocol in a conflict-free consistent way so that there is no single point of failure. With this hierarchy design, we are able to achieve high availability.

Implementation

DAG API

As Figure 3 shows, an application is a DAG of processors. Each processor handles messages.

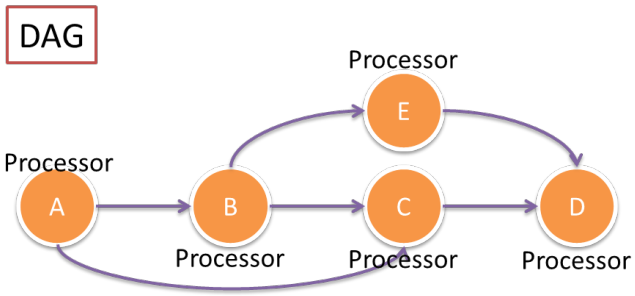


Figure 3: *Processor DAG*

At runtime, each processor can be parallelized to a list of tasks; each task is an actor within the data pipeline. Data is passed between tasks. The Partitioner (Figure 4) defines how data is transferred.

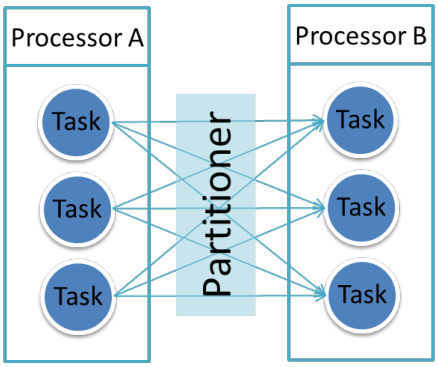


Figure 4: *Task Data Shuffling*

We have an easy syntax to express the computation. To represent the DAG above, we define the computation as:

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

A~>B~>C~>D is a Path of Processors, and *Graph* is composed of this Path list. This representation should be cleaner than that of vertex and edge set. A high-level domain specific language (DSL)

using combinators like flatmap will be defined on top of this. Our aim is to extend the existing Scala collections framework... likely by pimping[8].

Application Submission

In GearPump, each application consists of an AppMaster and a set of Executors, and each one is a standalone JVM. Thus, an application is isolated from other applications by JVM isolation.

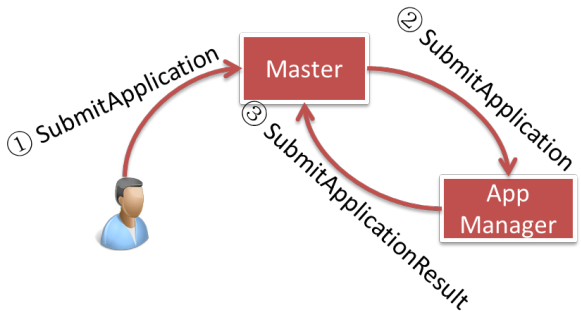


Figure 5: *User Submit Application*

To submit an application (Figure 5), a GearPump client specifies a computation defined within a DAG and submits this to an active master. The SubmitApplication message is sent to the Master, which then forwards it to an AppManager.

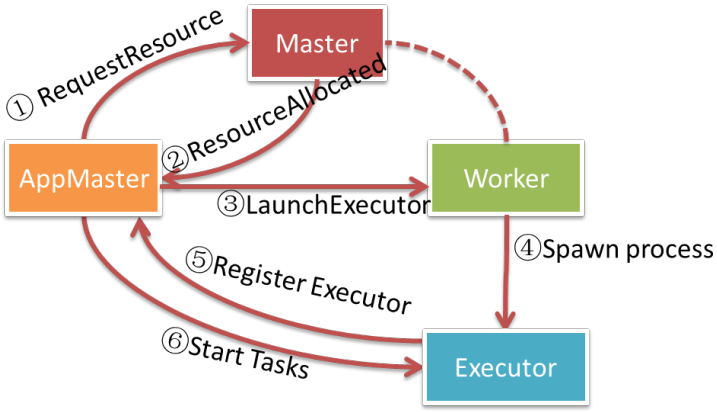


Figure 6: *Launch Executors and Tasks*

The AppManager locates an available worker and launches an AppMaster in a sub-process JVM of the worker. As shown in Figure 6, the AppMaster then negotiates with the Master for resource allocation in order to distribute the DAG as defined within the application. The allocated workers then launch Executors (new JVMs) and subsequently start Tasks.

Remote Actor Code Provisioning

Akka currently does not support remote code provisioning. That is, instantiating an actor on a remote node assumes this actor class and related dependencies are already available and accessible by the JVM's system class loader. GearPump provides several mechanisms that allow new resources to be loaded at a remote node site. This is a prerequisite to enabling run-time DAG modification

since you cannot assume new DAG sub-graphs have the requisite resources on the remote node. As noted above, when a worker launches an executor, the worker may modify the executor's classpath in several specific ways. First, it looks for any serialized jar sent as part of the application definition. If one exists, the worker saves this locally and appends the jar name to the executor's classpath. Second, the worker looks at the configuration settings also included as part of an application definition that may define an extra classpath for the executor. In this latter scenario resources are resident on the remote node but need to be appended to the executor's classpath. For example, an AppMaster and its task actors may need to access various Hadoop components during runtime that have already been preinstalled on the worker's machine.

High performance Message passing

For streaming applications, message-passing performance is extremely important. For example, one streaming platform may need to process millions of messages per second with millisecond level latency. High throughput and low latency are not that easy to achieve. There are a number of challenges:

1. First Challenge: Network is not efficient for small messages

In streaming, typical message size is very small, usually less than 100 bytes per message, like the floating car GPS data. But network efficiency is very bad when transferring small messages. As you can see in Figure 7, when message size is 50 bytes, it can only use 20% bandwidth. How to improve the throughput?

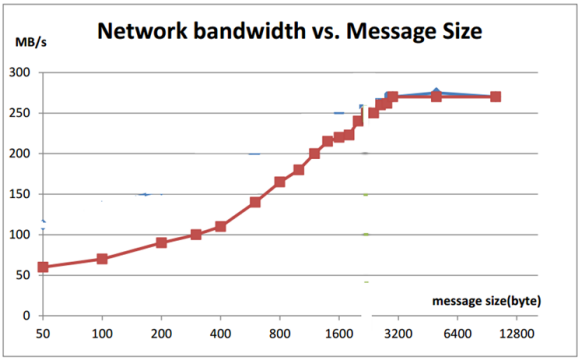


Figure 7: *Messaging Throughput vs Message Size*

2. Second Challenge: Message overhead is too big

Each message sent between two actors contains sender and receiver actor paths. When sending over the wire, the overhead of this ActorPath is not trivial. For example, the actor path below takes more than 200 bytes.

```
akka.tcp://system1@192.168.1.53:51582/remote/akka.tcp/2120193a-e10b-474e-bccb-8ebc4b3a0247@192.168.1.53:48948/remote/akka.tcp/system2@192.168.1.54:43676/user/master/Worker1/app_0_executor_0/group_1_task_0#-768886794
```

How do we solve these challenges?

We've implemented a custom Netty transportation layer with Akka extension. In Figure 8, Netty Client translates ActorPath to TaskId, and Netty Server translates it back. Only TaskId is passed on the wire. It is only about 10 bytes, so the overhead is minimized. Different Netty Client Actors are isolated; they do not block each other.

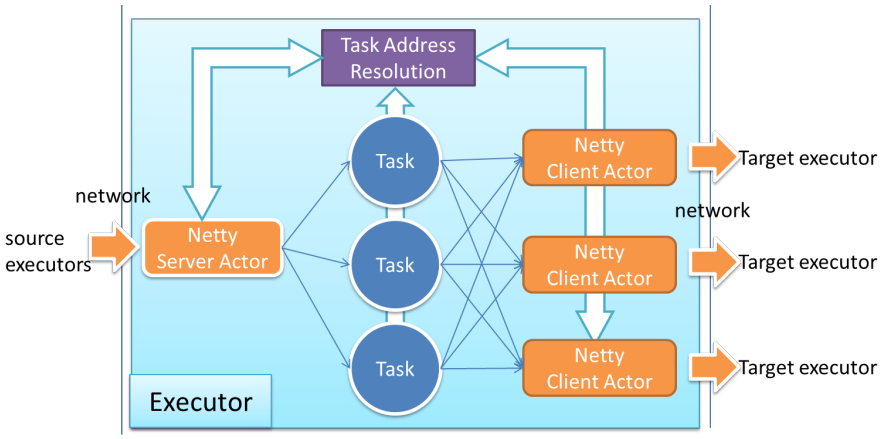


Figure 8: *Netty Transport and Task Addressing*

For performance, effective batching is really the key! We group multiple messages into a single batch and send it on the wire. The batch size is not fixed; it is adjusted dynamically based on network status. If the network is available, we flush pending messages immediately without waiting; otherwise, we put the message in a batch and trigger a timer to flush the batch later.

Flow Control

Without flow control, one task can easily flood another task with too many messages, causing out of memory errors. Typical flow control uses a TCP-like sliding window, so that source and target can run concurrently without blocking each other.

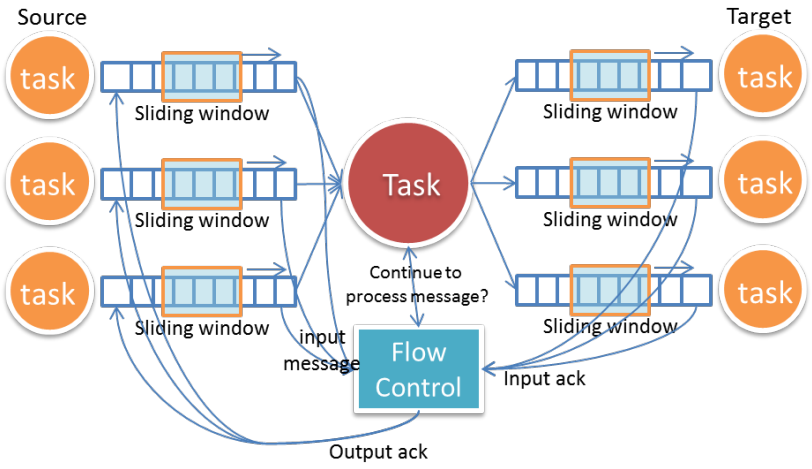


Figure 9: *Flow control, each task is "star" connected to input and output tasks*

The difficult part for our problem is that each task can have multiple input and output tasks. The input and output must be entangled so that the backpressure can be properly propagated from downstream to upstream, as shown in Figure 9. The flow control also needs to consider failures, and it needs to be able to recover when there is message loss.

Another challenge is that the overhead of flow control messages may be big. If we ack every message, a huge amount of ack'd messages will be in the system, degrading streaming performance. The approach we adopted is to use explicit AckRequest messages. The target tasks are only ack'd when they receive the AckRequest message, and the source only sends an AckRequest when necessary. With this approach, we largely reduced the overhead.

Message Loss Detection

The streaming platform needs to effectively track what messages have been lost and recover as fast as possible. For example, for web ads, the advertiser is charged for every ad click, so miscounts are not acceptable.

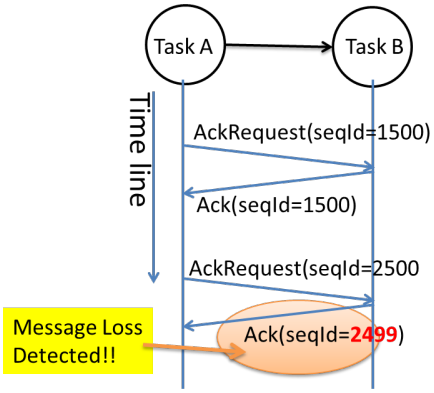


Figure 10: *Message Loss Detection*

We use the flow control message AckRequest and Ack to detect message loss (Figure 10). The target task counts how many messages have been received since the last AckRequest, and acks the count back to the source task. The source task checks the count and finds the message loss.

Stream Replay

In some applications, a message cannot be lost and must be replayed. For example, during a money transfer, the bank sends an SMS to us with a verification code. If that message is lost, the system must send another one so that money transfer can continue. We made the decision to use **source end message storage** and **timestamp-based replay**.

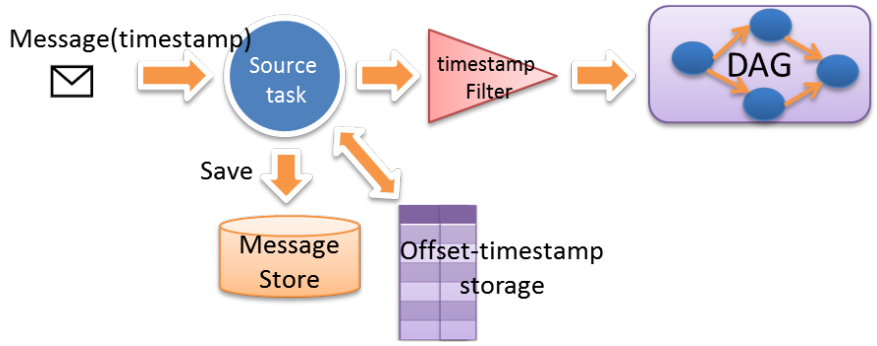


Figure 11: *Replay with Source End Message Store*

Every message is immutable and tagged with a timestamp. We made an assumption that the timestamp is approximately incremental (allow small ratio message disorder).

We assume the message is coming from a replayable source, like an Apache Kafka queue; otherwise, the message is stored at a customizable source end "message store". In Figure 11, when the source task sends the message downstream, the timestamp and offset of the message are also check-pointed to offset-timestamp storage periodically. During recovery, the system first retrieves the right timestamp and offset from the offset-timestamp storage. Then it replays the message store from that timestamp and offset. A Timestamp Filter filters out old messages in case the message in storage is not strictly time-ordered.

Master High Availability

In a distributed streaming system, any part can fail. The system must stay responsive and do recovery in case of errors.

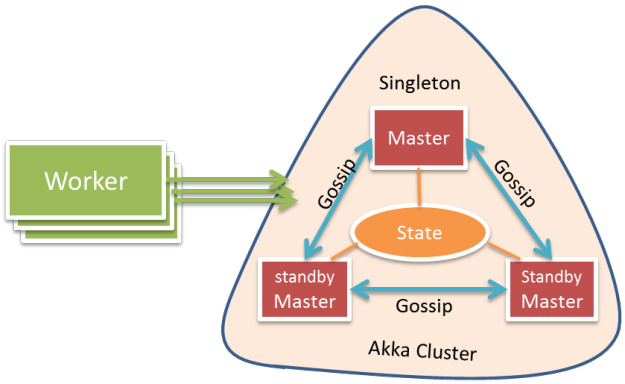


Figure 12: *Master High Availability*

We use Akka clustering, as shown in Figure 12, to implement the Master high availability. The cluster consists of several master nodes, but no worker nodes. With clustering facilities, we can easily detect and handle a master node crash. The master state is replicated on all master nodes with the Typesafe akka-data-replication[9] library. When one master node crashes, another standby master reads the master state and takes over. The master state contains the submission data of all applications. If one application dies, a master can use that state to recover that application. CRDT LwwMap[10] is used to represent the state; it is a hash map that can converge on distributed nodes

without conflict. To have strong data consistency, the state read and write must happen on a quorum of master nodes.

Global Clock Service

Global clock service tracks the minimum timestamp of all pending messages in the system. Every task updates its own minimum-clock to global clock service (Figure 13); the minimum-clock of a task is decided by the minimum of:

- Minimum timestamp of all pending messages in the inbox.
- Minimum timestamp of all unacked outgoing messages. When there is message loss, the minimum clock does not advance.
- Minimum clock of all task states. If the state is accumulated by a lot of input messages, then the clock value is decided by the oldest message's timestamp. The state clock advances by doing snapshots to persistent storage or by fading out the effect of old messages.

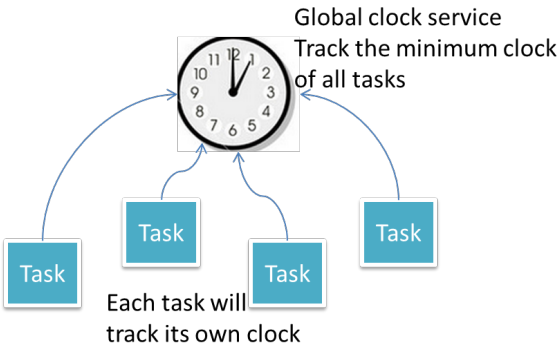


Figure 13: *Global Clock Service*

The global clock service keeps track of all task minimum clocks and maintains a global view of minimum clock. The global minimum clock value is monotonically increasing, meaning that all source messages before this clock value have been processed. If there is message loss or task crash, the global minimum clock stops.

Fault Tolerant and At-Least-Once Message Delivery

With Akka's powerful actor supervision, we can implement a resilient system relatively easily. In GearPump, different applications have a different AppMaster instance and are totally isolated from each other. For each application, there is a supervision tree, AppMaster->Executor->Task. With this supervision hierarchy, we can free ourselves from the headache of zombie processes. For example if AppMaster is down its Akka supervisor ensures the whole tree shuts down.

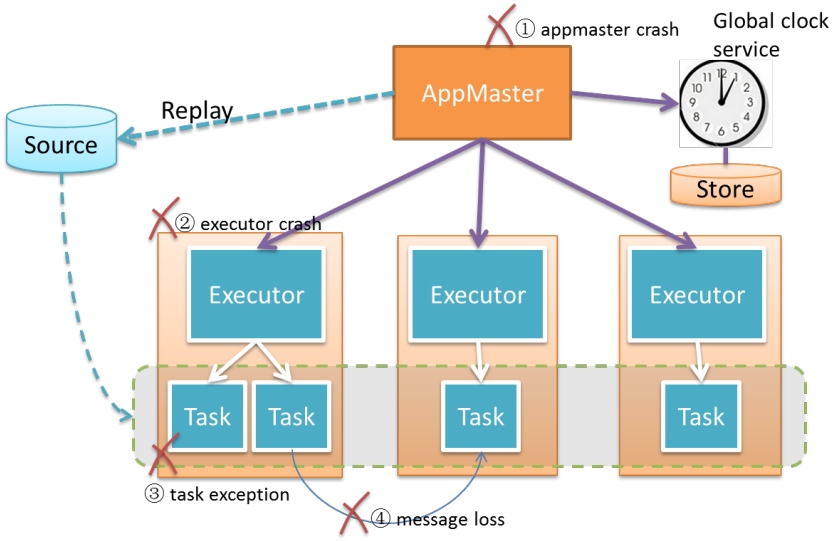


Figure 14: *Possible Failure Scenarios and Error Supervision Hierarchy*

It is also easier to handle message loss and other failures. Multiple failure scenarios are possible, as shown in Figure 14. If an AppMaster crashes, Master schedules a new resource to restart the AppMaster, and then the AppMaster requests resources to start all executors and tasks in the tree. If an Executor crashes, its supervisor AppMaster is notified and requests a new resource from the active master to start the executor's child tasks. If a task throws an exception, its supervisor executor restarts that Task.

When "at least once" message delivery is a requirement, we replay the messages in the case of message loss. First, AppMaster reads the latest minimum clock from the global clock service or clock storage if the clock service crashes. Then AppMaster restarts all the task actors to get a fresh task state, and the message source replays messages from that minimum clock. This can be further improved to restart partial tasks only.

Exactly Once Message Delivery (ongoing)

For some applications, it is extremely important to do "exactly once" message delivery. For example, for a real-time billing system, we do not want to bill the customer twice. The goal of exactly once message delivery is to make sure:

- Errors don't accumulate—today's error will not be accumulated to tomorrow
- State management is transparent to application developer

We use global clock to synchronize the distributed transactions. We assume every message from the data source has a unique timestamp. The timestamp can be a part of the message body or can be attached later with system clock when the message is injected into the streaming system. With this global synchronized clock, we can coordinate all tasks to checkpoint at the same timestamp.

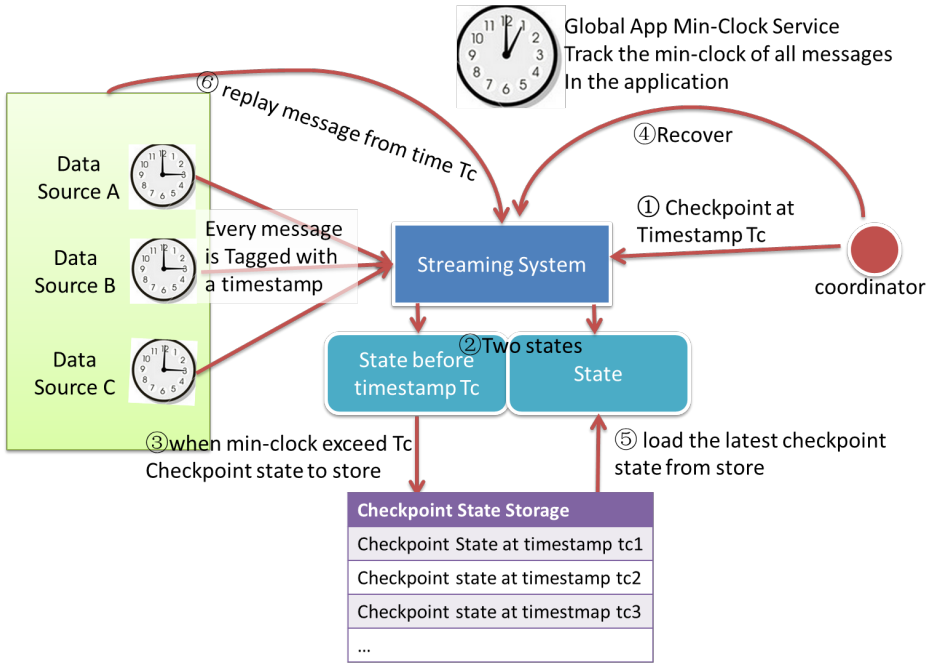


Figure 15: *Checkpointing and Exactly Once Message delivery*

Workflow to do state checkpointing (Figure 15):

1. The coordinator asks the streaming system to do a checkpoint at timestamp Tc.
2. Each application task maintains two states: checkpoint state and current state. Checkpoint state only contains information before timestamp Tc. Current state contains all information.
3. When the global minimum clock is larger than Tc, all messages older than Tc have been processed. The checkpoint state will no longer change, so we can then safely persist the checkpoint state in storage.
4. When there is message loss, we start the recovery process.
5. To recover, load the latest checkpoint state from storage and use it to restore the application status.
6. Data source replays messages from the checkpoint timestamp.

The checkpoint interval is dynamically determined by the global clock service (Figure 16). Each data source tracks the max timestamp of input messages. Upon receiving min clock updates, the data source reports the time delta back to global clock service. The max time delta is the upper bound of the application state timespan. The checkpoint interval is bigger than max delta time as determined by:

$$\text{checkpointInterval} = 2^{1+\lfloor \log_2 \max\_delta\_time \rfloor}$$

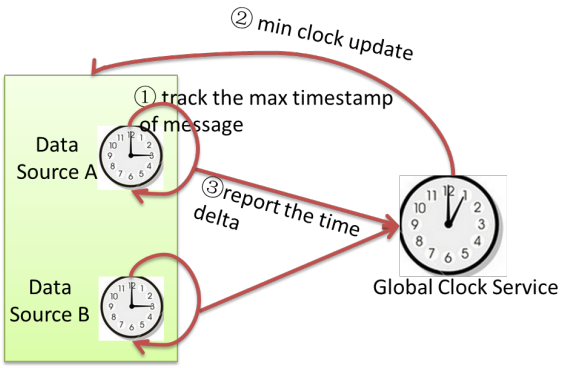


Figure 16: *How to determine Checkpoint Interval*

After the checkpoint interval is notified to tasks by the global clock service, each task calculates its next checkpoint timestamp autonomously without global synchronization.

$$\text{nextCheckpointTimeStamp} = \text{checkpointInterval} \cdot \left(1 + \left\lfloor \frac{maxMessageTimestampOfThisTask}{checkpointInterval} \right\rfloor\right)$$

Each task contains two states: checkpoint state and current state. The code to update the state is shown in the listing below (List 1).

```
TaskState(stateStore, initialTimeStamp):
  currentState = stateStore.load(initialTimeStamp)
  checkpointState = currentState.clone
  checkpointTimestamp = nextCheckpointTimeStamp(initialTimeStamp)
onMessage(msg):
  if (msg.timestamp < checkpointTimestamp):
    checkpointState.updateMessage(msg)
  currentState.updateMessage(msg)
  maxClock = max(maxClock, msg.timeStamp)

onMinClock(minClock):
  if (minClock > checkpointTimestamp):
    stateStore.persist(checkpointState)
    checkpointTimeStamp = nextCheckpointTimeStamp(maxClock)
    checkpointState = currentState.clone

onNewCheckpointInterval(newStep):
  step = newStep
nextCheckpointTimeStamp(timestamp):
  checkpointTimestamp = (1 + timestamp/step) * step
```

List 1: Task Transactional State Implementation

The in-memory current state can be queried directly by the client, it represents the real time view of the data.

Dynamic Graph (ongoing)

We want to be able to modify the DAG dynamically so we can add, remove, and replace sub-graphs (Figure 17).

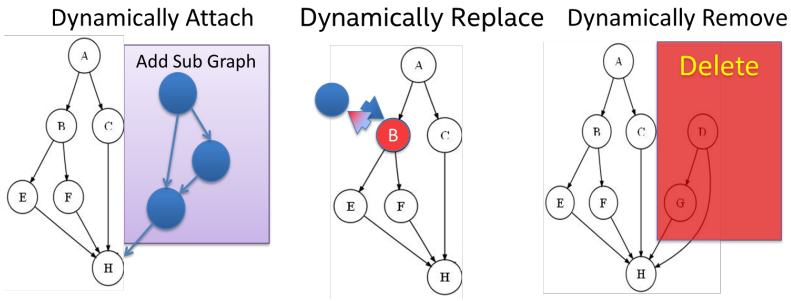


Figure 17: *Dynamic Graph, Attach, Replace, and Remove*

Observations on Akka

Akka is a great tool for streaming. The message-driven architecture and intuitive Actor Model largely influence how a distributed service should be designed. Akka and its libraries give us enough flexibility and power to build a real-time and scalable distributed streaming platform.

- With Akka remoting, we can easily provision a computation DAG of remote nodes, including code provisioning.
- With Akka clustering, we immediately have a distributed consistent state to build a high available service.
- With location transparency, we can handle scale-up and scale-out in the same way; it is also easier for us to test our code using a single node.
- With an error supervision tree, we are better situated to handle the errors and recover with the best approach.
- Akka has very good extensibility; we have customized Akka on serialization, message passing, and metrics to meet particular needs.

Because streaming applications has extreme demands on throughput and latency when processing large volumes of data, we customized Akka to better meet our performance requirements.

- For the message passing between tasks, we implemented a customized message passing extension that batches the messages before sending across the network.
- To reduce the actor path overhead, we developed a lightweight method to address the task uniquely in the application.
- To reduce the CPU time of serialization, we avoid using a pooled serializer and the heavy serializer lookup process. We also avoid serializing the same message multiple times.
- To reduce the Akka clustering messaging overhead, the worker is designed to NOT be part of Akka cluster.

- For flow control, our implementation is slightly different with Akka streaming, the rate of Ack messages is smaller.

One case we have not solved is timeout detection. We want to avoid the following two scenarios:

A) When one node crashes, the watching node takes too long to note its death.
B) When the CPU load is high, actors may start to disassociate with other nodes.

The reality is that either or both of these may occur. We want a faster and more robust way to differentiate real failures from temporary unavailability, so we can recover faster in case of real failures and be more resilient in case of temporary load increases.

Experiment Results

To illustrate the performance of GearPump, we mainly focused on two aspects: throughput and latency, using a micro benchmark called SOL (an example in the GearPump distro) whose topology is quite simple. SOLStreamProducer delivers messages to SOLStreamProcessor constantly and SOLStreamProcessor does nothing. We set up a 4-node cluster (each node was an Intel® Xeon® 32-core processor E5-2680 2.70GHz with 128GB memory) and a 10GbE network.

Throughput

Figure 18 shows that the whole throughput of the cluster can reach about 11 million messages/second (100 bytes per message).

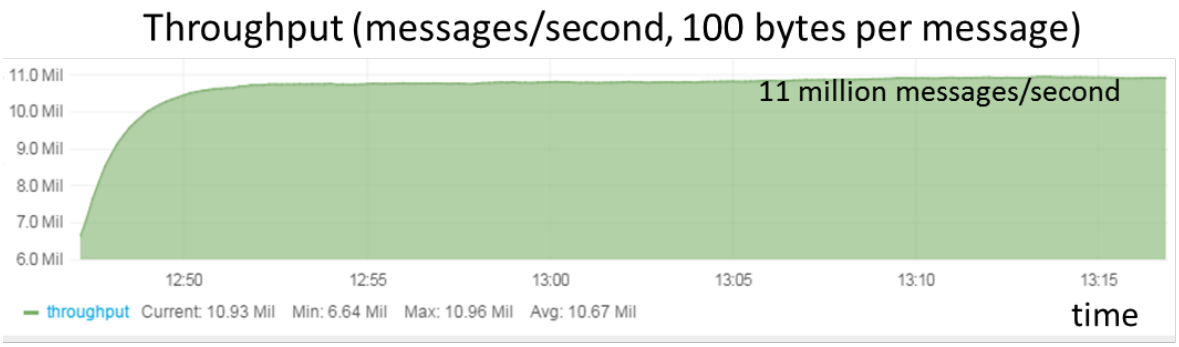


Figure 18: *Performance Evaluation, Throughput, and Latency*

Latency

When we transfer messages at the max throughput above, the average latency (Figure 19) between two tasks is 17ms with a standard deviation of 13ms.

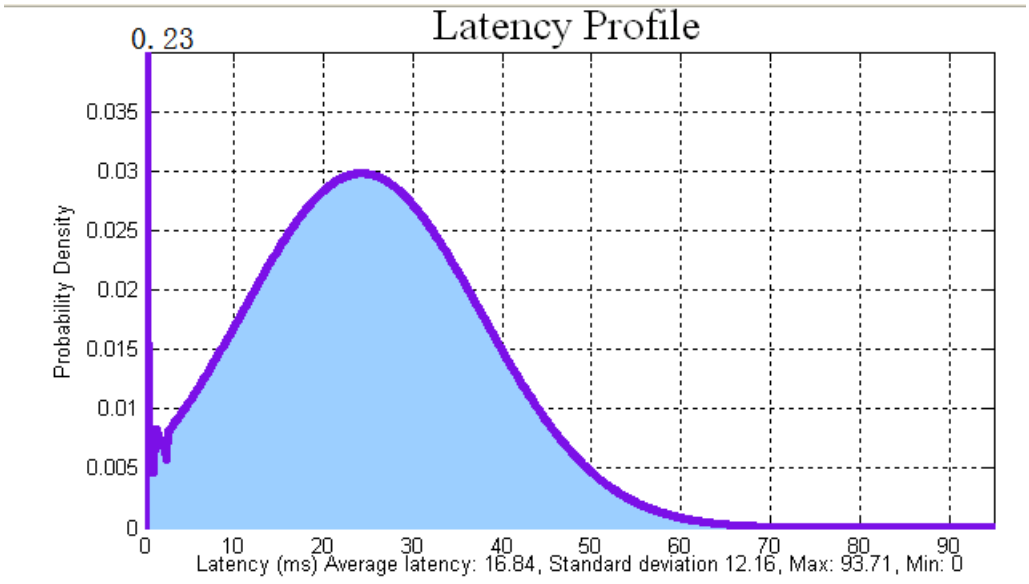


Figure 19: *Latency between Two tasks(ms)*

Fault Recovery time

When corruption is detected—for example, the Executor is down—GearPump reallocates the resource and restarts the application. It takes about 10 seconds to recover the application.

Future Plan(s)

We are still working on transactional support and dynamic graph support for our next release. Besides these two, we also want to incorporate these features:

- StreamSQL
  Per O'Reilly Media Strata's report, *Data Science Salary Survey* in 2013， SQL is the most used data tool for data scientists. StreamSQL is a must-have feature. We want to support translating a SQL statement to a running DAG, and supporting stream special functions like windows operations.
- Visualization
  Everything is moving in streaming applications, and it is hard to know when unexpected things happen, so visualization is extremely important. Visualization will give us insights about how the computation DAG is composed, latency and throughput, system and network load status, and error information. We have preliminary REST support that emits a given AppMasters DAG in JSON. We'll likely have visualization using *viz.js* in the next release.
- Data skew handling with Auto-Split
  Data skew is common in big data, and it highly impacts application performance. In case of data skew, some tasks may be too busy, while the others are free. We want the busy task to be able to split into small tasks automatically in response to load change.

Summary

Akka is characterized by its simplicity and elegance. The concepts of being message driven and providing computation isolation makes Akka a natural fit for streaming frameworks. Akka provides

a solid foundation that allows developers to focus on the data logic and the specific requirements of streaming. GearPump exploits the current Akka feature set and is small and nimble, only 5000 lines of code in total yet provides compelling streaming features including scalability, low latency, resilience, error isolation, and high availability. Performance is not traded for features: 11 million messages per second is a strong performance metric. Finally, we provide a growing number of innovations including code provisioning and in-flight DAG modification. The platform is rapidly evolving and will be directed towards solving modern day challenges at scale. We invite those interested to explore the tutorial at https://github.com/intel-hadoop/gearpump.

Reference

[1] Millwheel http://research.google.com/pubs/pub41378.html
[2] Apache Storm http://storm.apache.org/
[3] Spark Streaming https://spark.apache.org/streaming/
[4] Apache Samza http://samza.incubator.apache.org/
[5] Apache TEZ http://zh.hortonworks.com/hadoop/tez/
[6] Hadoop YARN http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[7] YARN Architecture Paper Section 3 http://dl.acm.org/citation.cfm?id=2523633
[8] Scala class implicits https://www.google.com/search?q=scala+pimp+my+library
[9] Akka Data replication library https://github.com/patriknw/akka-data-replication
[10] A comprehensive study of Convergent and Commutative Replicated Data Types https://hal.archives-ouvertes.fr/file/index/docid/555588/filename/techreport.pdf

Notices

cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.