Telemetry for Elastic Data (TED): Middleware for MapReduce Job Metering and Rating

Soheil Qanbari*, Ashkan Farivarmoheb†, Parham Fazlali†, Samira Mahdi Zadeh*,and Schahram Dustdar*
*Distributed Systems Group, Vienna University of Technology, Vienna, Austria
{qanbari, dustdar}@dsg.tuwien.ac.at
e1329639@student.tuwien.ac.at
†Baha'i Institute for Higher Education (BIHE)
{ashkan.farivar, parham.fazlali}@bihe.org

***Abstract*—The consumption-based rating of MapReduce jobs is tightly coupled with metering the infrastructure resource usage it runs on. In this context, metering and controlling the job execution depends on the number and type of containers used to setup and run the Hadoop cluster as well as the duration of the job execution. Duration-basis metering like an hourly rate for every instance per hour usage, poses challenges of surcharge of jobs lasting less/more than an hour. Jobs lasting for 61 minutes will unfairly be charged for two hours. In response to these findings, the authors offer *Job-basis* telemetry mechanism rather than *Duration-basis* where the metering granularity is carried on MapReduce DAG bundles, jobs and tasks levels. This model is developed as an elastic data telemetry (TED) middleware to provide real-time resource utilization awareness over data-intensive applications. Clients will benefit from this model by enforcing their applications elasticity policies and achieve pricing transparency over their actual usage. This granular elasticity control is achieved by moving jobs among priority queues which fit cost and quality requirements. TED collects the emitted usage data stream, generates billable artifacts to form a *tailored* policy (scale up/down) to satisfy several desirable properties. This contributes to a supervised, finer-grained resource allocation due to the application behavior.**

I. Introduction

Utility computing[1], [2] is an evolving facet of cloud computing that aims to leverage and treat computing resources as a metered service, like natural gas. It enables a *Pay-per-use* or *utility-based* pricing model through *metered data* to achieve more financial transparency. Metering measures rates of resource utilization via metrics, such as data storage or memory usage, consumed by the cloud service subscribers. Metrics are statistical units that indicate how consumption is measured and priced. Furthermore, metering is the process of measuring and recording the usage of an entire application, individual parts of an application, or specific services, tasks and resources. From the provider view, the metering mechanisms for service usage differ widely, due to their offerings that are influenced by their cloud business models. Such mechanisms range from usage over time, volume-basis to subscription models. Thus, providers are encouraged to offer reasonable pricing models[3] to monetize the corresponding metering model.

In the cloud market, providers are expected to have a layered metering model together with its associated pricing schema to exploit various resource usage granularity. In effect, the more fine-grained the metering service is, the more transparent is the utilization. Data aggregation is needed to provide a broader view of resource usage and conversely, small-footprint or micro metering is required to achieve a more granular view of the resource utilization. Thus, increasing the resolution and precision of metering the underlying resource unit to improve the validity of the quantified cost appears to be vital. However, this comes with an elevated cost for monitoring, which we assume is reasonable. In this context, the underlying resource usage events are time-series data that are streamed at tiny time intervals (e.g., 3 seconds) to enable current and consistent data retrieval of a metered unit.

The quest for telemetry of the client's job resource usage becomes more challenging when the job is deployed and processed in a distributed model. For instance, the MapReduce framework[4] offers an abstraction that simplifies the execution of data processing. Such computation intensive applications run in a distributed setting while hiding the details of parallelization, data distribution, load balancing and fault tolerance. It aims to parallelize the data-intensive application processing and less focus is put on efficient underlying resource utilization. Enriching MapReduce with telemetry services will contribute to more optimized resource utilization and performance in the cluster. This leads to a question, how can granular metering contribute to more utilization value? The reason is granular elasticity control. Metering *Map* or *Reduce* tasks enables granular elasticity control on multiple levels by varying elasticity requirements like cost and quality[5]. The next question is how does granular metering improve performance? The solution is process-weight classification of the application jobs. With the classification framework, the job types are classified in terms of resource usage (map or reduce-intensive), and this seeds algorithms for an optimal tailored scheduling. This leads to a more optimized resource allocation than other current plugged policies. From the consumer view, clients seek to economize their usage patterns by optimizing their map or reduce-intensive jobs. Along with these motivations, MapReduce-based utility computing contributes to planning a cluster's future resource requirements for more elasticity and performance. The flip side of the coin is that providers will gain an understanding of how their underlying resources are being consumed to bill users respectively. Thus, it makes eminent sense to meter emitted data of MapReduce resource usage.

Cloud Market-Leader Amazon offers an *Elastic MapReduce (EMR)* web service, for instance, a hosted platform on the Amazon cloud where users can instantly provision Hadoop clusters to perform their data-intensive tasks. Amazon EMR uses Hadoop, an open source framework, to distribute your

978-1-4673-7952-6/15 $31.00 © 2015 IEEE
DOI 10.1109/Trustcom.2015.568

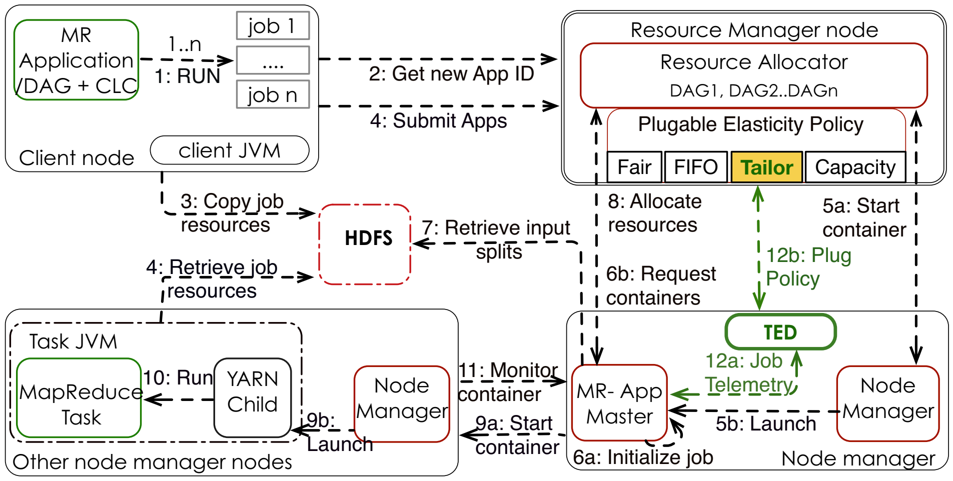

Fig. 1: TED middleware data and process flow with their sequence of events in YARN

data and processing across a resizable cluster of Amazon EC2 instances. Based on the Amazon EMR pricing[1] model, you pay an hourly rate for every instance-hour you use. The hourly rate depends on the instance type used (e.g. standard, high cpu, high memory, high storage, etc). Hourly prices range from $\$0.011/hour$ to $\$0.27/hour$ ($\$94/year\$ to$ $2367/year$). The Amazon EMR price is in addition to the Amazon EC2 price (the price for the underlying servers). There are a variety of Amazon EC2 pricing options, including *On-demand*, *Reserved*, and *Spot* instances. Besides, Amazon EMR uses other services such as Amazon S3, SQS, SimpleDB for its operations which are billed separately.

Due to the Amazon EMR pricing model, a 10-node cluster running for 10 hours costs the same as a 100-node cluster running for 1 hour. Kambatla et al [13] drilled this down for more transparency and showed us that resource elasticity for MapReduce jobs is not entirely symmetric, *i.e.,* 1 hour on 100 nodes may not accomplish the same resource usage throughput as 100 hours on 1 node. This poses a challenging decision of choosing the right cost-efficient cluster size. Following the previous scenario, there is a potential situation where the duration of job's execution time comes into play. If the job doesn't go into overtime, but runs for 61 minutes then you would get charged for the next hour, doubling the cost of the job! The most remarkable feature of EMR billing is that Amazon bills by an hourly increment. Cluster initialization seems to take 5 minutes or so with the Amazon distribution, so if one of your job's input paths is missing, it will take about 5 minutes to fail, but you'll be charged for the whole hour. You can definitely buy more machines and get a job done in less than an hour, but the cost-efficiency goes down. For instance, if your job completes in 30 minutes you will roughly double the cost.

However, in this context, the visible challenge for MapReduce jobs that must scale to extremely high capacity, is to ensure actual application resource consumption in terms of quality and cost elasticity. To address the above challenges, we implement an elastic data telemetry middleware called TED for YARN framework. TED collects "actual telemetry" events of running map/reduce tasks and then "meters" such data in meaningful way by returning the outcomes of its granular metering for two purposes. First, to generate billing statements to charge clients respectively and, second to generate and enforce tailored policies to control current applications' resource consumption behavior.

A mapreduce application can be represented by a directed acyclic graph (DAG). The DAG is used to represent a set of tasks where the input, output, or execution of one or more tasks is dependent on one or more other jobs. The tasks are nodes in the graph, and the edges identify the dependencies. The elasticity strategies can be applied on the whole DAG or parts of it. Similarly a set of DAGs can be bundled into a data analytic workflow driven by the Apache Oozie engine[2]. Some strategies might constitute constraints over the number of pending or running DAG jobs, maps or reduces.

As illustrated in figure 1, we have positioned and deployed our TED middleware in the resource manager node on the YARN architecture. The figure also depicts the event flow sequence corresponding to the mapreduce job life-cycle together with its metering process. YARN dynamically allocates resources for MR Jobs as they run. Let us demonstrate how TED utilizes resource provisioning. In this scenario, at stages $1, 2, 3, 4$ as shown in Fig. 1, a client submits a job or a DAG of jobs with the required container launch context (CLC) information to the resource manager and copy data source to the HDFS. As soon as the job is submitted, the *ResourceAllocator* negotiates a container and instantiates the *ApplicationMaster* for the job at stages $5a$ and $5b$. At this moment, stages $6a$ and $6b$, the *ApplicationMaster* initializes the job and requests containers from the *ResourceManager*. Then, at $9a$, $9b$, 10 and 11, it launches the granted container, runs the job and monitors its execution. The containers are configured based on the CLC specification. Our focus lies on stages $12a$ and $12b$, where TED retrieves and meters the resource usage data, interprets the job's behavior and then plugs the suitable and tailored

[1]http://aws.amazon.com/elasticmapreduce/pricing/

[2]http://oozie.apache.org

elasticity policy (scale-up/down) into the resource allocator for enforcement.

ApplicationMaster monitors the job execution flow through NodeManager, and observes their status and resource usage metrics (cpu, memory, disk, network). Then it negotiates resources for a single application (a single job or a directed acyclic graph of jobs). The ApplicationMaster initial request is structured in the `[resource-name, priority, resource-requirement, number-of-containers]` format. Then, via heartbeats negotiates ResourceManager its changing resource needs. After negotiation, it applies the dynamic adjustments to ensure resource consumption restraints are met. ResourceManager controls the container allocation by enforcing our plugged elasticity policy for a specific job. Before launching the container it has to construct the CLC object according to its needs which can include the allocated resource capability, security tokens, resource dependencies, environment variables, local directories. Next is to execute the task on the launched container.

To this end, our contribution is twofold: (i) An elastic and granular metering and resource scheduling mechanism for a class of MapReduce-based applications. (ii) In support of such a mechanism, we develop an elastic data telemetry (TED) framework as a basis for a multi-level resource metering model. This contributes to fine-grained resource consumption transparency and elasticity control. Our TED framework implements a layered approach to the interpretation of the time series resource usage events. TED achieves resource granular metering model in a hierarchical modeling topology. Having the metered data collected, TED highlights operational events, aggregates, and enriches them with the corresponding pricing model, then correlates the data with the associated client account and generates the billable artifact respectively. This leads to useful insights into the MapReduce job behavior for generating new allocation policies to achieve the intended performance defined in the service level agreements (SLAs).

With this motivation in mind, the paper continues with an initial analysis to identify and elicit the requirements for the TED framework in section II. With some definitive clues on how the MapReduce jobs are currently being metered and monetized, we propose a new elastic data telemetry framework (TED) to fulfill each requirement. The TED framework layered architecture together with its interacting components are detailed in section III. This section is devoted to the core elements of the TED model *i.e.,* resource usage retrieval, MapReduce metering metrics, pricing kernel, a classification model for granular usage pattern interpretation, and a billing gateway to expose telemetry billable artifacts via FIX protocol. In support of our model, we have developed a primary TED framework able to handle MapReduce job metering in a Hadoop YARN cluster. The available prototype[3] is put to production by metering real MapReduce jobs. We evaluate our TED framework and numerical results will be given in section IV to prove the efficiency of our model. Subsequently, section V surveys related work. Finally, section VI concludes the paper and presents an outlook on future research directions.

[3]https://github.com/soheil4TUWien/TED

II. Resource Consumption Metering Requirements

In architecting our metering middleware we initially defined the requirements in which it will operate. The appropriate metrics for the metering methods and the jobs that are metered could vary quite significantly based on factors such as telemetry requirements and application domain context. Next, we elicit our requirements.

A. YARN Cluster Capacity Planning via Metering

YARN clusters are elastic in nature. They host MapReduce applications and adapt themselves to varying loads by elastically allocating and releasing underlying resources to map and reduce tasks. Obviously, it is an obligation for YARN providers to assure the availability of required supply. Otherwise, lack of resources results in unmet demands and poor performance, triggering the SLA violations and leads to financial consequences and penalties. YARN supply planning depends on a few factors: types of machines (Nodes), types of workload (Memory/Storage/CPU-intensive), Number of tasks (map or reduce) per node and etc. Usually count 1 core per task. If the job is not too heavy on the CPU, then the number of tasks can be greater than the number of cores. For instance: 12 cores, jobs use 75% of CPU, free task slots = 14, maxMapTasks = 8, maxReduceTasks = 6. By default, the tasktracker and datanode take each 1GB of RAM. For each task calculate `mapred.child.java.opts` (200MB per default) of RAM. In addition, count 2GB for the OS. So say, using 24GB of memory, $24 - 2 = 22$GB available for our tasks – thus we can assign 1.5GB for each of our 14 tasks ($14 * 1.5 = 21$GB). YARN uses `yarn.nodemanager.resource.memory-mb` and `yarn.nodemanager.resource.cpu-vcores` to control the amount of *memory* and *cpu* on each node, which both are available to *maps* and *reduces*. Resource allocation plans and elasticity requirements can be enforced using these configurations.

Metering can provide valuable information about these factors to show the way that an application is used. This requires classifying the job types (CPU bound, Memory or Disk I/O bound, or Network I/O bound) into: balanced workload, compute intensive, I/O intensive or evolving workload patterns. Such classification can identify trends that indicate future needs such as storage and compute resource requirements. Moreover, this may indicate which resources are more utilized in map or reduce intensive job phases affecting response times. Such utilization knowledge might also influence the effort towards development of optimized storage methods, cost reduction or additional storage, like using AWS spot instances[4]. The fundamental unit of resource allocation in YARN is the *priority queue* which we will discuss later in details.

B. Metering for Granular Consumption Transparency

The level of a resource metering unit becomes an essential facet of facilitating a way of hierarchical metering and processing of usage patterns indexed by information granules. By granule metering, we mean a collection of metrics aggregated together by their functional relationship or closeness. Such

[4]aws.amazon.com/ec2/purchasing-options/spot-instances/

granules are then formulated by adopting and leveraging a certain level of abstraction to achieve further utility. Each abstraction level is formed by grouping metrics together into semantically meaningful constructs to reflect the structure of the original data into its granular counterpart. Granular metering enables diving deeper into measuring the resource usage on the *per-DAG-flow*, *per-DAG*, *per-Job*, *per-Map* or *per-Reduce* levels. Such metering granules can be regarded as more abstract and interpretable entities in charging clients and in elasticity policy enforcement. We treat them as a scale unit. This provides users real-time visibility over their resource consumption and the ongoing money stream being paid as they go. Furthermore, it enables clients realtime application control to ensure that quality and cost constraints are met.

Meanwhile, we see the resource usage events as time-series data whereas the consumption information granulation occurs in time intervals. In this context, the duration of the map or reduce phases' run could form a scale of time granulation. As elaborated above, we identify three factors: *granule unit*, *time interval* and *metric type* to building granular representatives of usage data. The granulation mechanism involves criterion of closeness of elements, and if required, could also embrace some aspects of functional resemblance. In other words: we collect underlying usage events and elevate them into granule units in reasonable time intervals. Such granular classification of application metering enable *Pay-per-granules* enables for mapreduce-based applications where customers are billed based on their actual application resource usage and can track their ongoing costs.

C. Fine-grained Metering for Elasticity Control

Elastic metering allows fine-grained adequate resource allocation and prevents exceeding the preset resource usage and limits. By elastic metering we mean to enforce resource quotas on the cost and quality constraints. This requires a scheduled utility that observes key threshold constraints and fires the appropriate notification event to enforce the suitable elasticity control policy like *Scale-in* or *Scale-out*. To apply such control, we provide a granular classification on the usage data. The schemes of granular classification are comprised of several functional steps. A crux of the scheme is shown in the following:

$$\text{Usage Events } (1)\xrightarrow[Retrieval]{\texttt{[key,value]}} \text{Metrics Feature Space}$$
$$(2)\xrightarrow[Aggregation]{Granulation} \text{Granular Feature Space } (3)\xrightarrow[Association]{Correlation}$$
$$\text{Usage Interpretation } (4)\xrightarrow[Enforcement]{TailoredPolicy} \text{Queue Classifier}$$

Let us briefly elaborate on the essence of the successive phases of the overall metering scheme. The first step is dedicated to meaningful representation of the resource usage data to form a metric feature space. As a result of this representation, one returns a vector of numeric descriptors characterizing the time series and used in consecutive phases. This vector of `jobTaskAttemptCounters` contains a list of `[key,value]` of our job metrics. In our case we retrieve the `CPU_MILLISECONDS`, `PHYSICAL_MEMORY_-BYTES` and `VIRTUAL_MEMORY_BYTES` metric values of tasks for further processing.

At any given time t in the second phase, TED provides enriched pieces of information about the MapReduce job resource usage observations. Such observations can be represented as a column-vector $\mathbf{o}_t \equiv [v_{t,1}\ v_{t,2}\ ...\ v_{t,n}]^T \in R^n$ of YARN metrics data stream values at time t. The stream of usage data can be regarded as a frequently expanding $t \times n$ matrix $\mathbf{O}_t := [\mathbf{o}_1\ \mathbf{o}_2\ ...\ \mathbf{o}_t]^T \in R^{t\times n}$ where the new incoming streams are added as matrix rows at each time interval t in real-time. In our YARN case, $\mathbf{O}_t$ is the measurements column-vector at t over all the metrics, where n is the length of the vector and indicates the number of hadoop metrics and t is the measurement time-stamp. These vectors represent the set of measurements obtained for the n metrics at a specific observation. In particular the rows of the matrix represent various `monitoring observations` in a given period, while the columns are the sample `values` detected for each metric during the observations.

$$\mathbf{O}_{t,n} = \begin{pmatrix} jobID_{1,1} & taskID_{1,2} & CPU_{1,3} & \cdots & Mem_{1,n} \\ jobID_{2,1} & taskID_{2,2} & CPU_{2,3} & \cdots & Mem_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ jobID_{t,1} & taskID_{t,2} & CPU_{t,3} & \cdots & Mem_{t,n} \end{pmatrix} \tag{1}$$

In our scenario, as shown in Equation 1, let **O** be a matrix representing the job's tracking data measured by metrics counters in the allocated container. For instance, `key` element of $CPU_{t,3}$ represents the CPU usage `value` of the specific $taskID_{t,2}$ of the specific $jobID_{t,1}$ at time *t*. This leads to granular representation of time series data.

In the third phase, a collection of information granules is constructed and positioned as belonging to granularity classes. This forms a layered approach to constructing a classification framework of granular interpretation of time-varying resource consumption events. Such interpretation abilities could help to understand the type and behavior of the job in terms of resource insensitivity. The queue classifiers positioned as the last functional module of a metering framework scheme are used to realize mapping of discovered running job types on the current job scheduling policy and its associated elasticity control strategy class labels. This mapping helps in choosing the suitable economic decisions and accordingly actions taken on the job elasticity control like moving the job to a proper queue.

Having such classifications in effect, TED manages the level of resource provisioning on various granules to a specific YARN deployment to keep the job up and running and ensure elasticity requirements are met. Next, we describe the design and implementation of our metering solution for mapreduce-based applications deployed in YARN cluster. Last, but not least, about the architecture robustness, if TED faces its own deployment issues then this may have a major impact on vendor profitability. Our approach is to schedule some background log analysis utilities like logstash[5] to detect and even restart the suspended TED instance.

[5]http://logstash.net

III. TED Framework Architecture

Looking forward, figure 2 provides a schematic view on architecting TED's collaborating components. The next section lays out the basis for the underlying resource metering metrics together with its retrieval mechanism.

A. *Usage Data Retrieval*

Collecting and streaming usage data from mapreduce jobs in Hadoop needs a lightweight solution to avoid additional network I/O for the sake of performance. Each usage event contains information about the job like subscriber, timing, the result (success, failure), resource usage metrics and their values. One solution is to receive notification events via callback feature of the Hadoop. At job completion, an HTTP request will be sent to `job.end.notification.url` value[6]. Both the `JOB_ID` and `JOB_STATUS` can be retrieved from the notification URL that we supplied in Job configuration. The URL connection is fire-and-forget (FAF)[7]. We take a skeptical view of this approach since we will not receive any notification in case of task completion. Typically, when you run a map/reduce job you get the object of type `org.apache.hadoop.mapreduce.Job`. Using this object you can poll the JobTracker in a predefined interval to check its status.

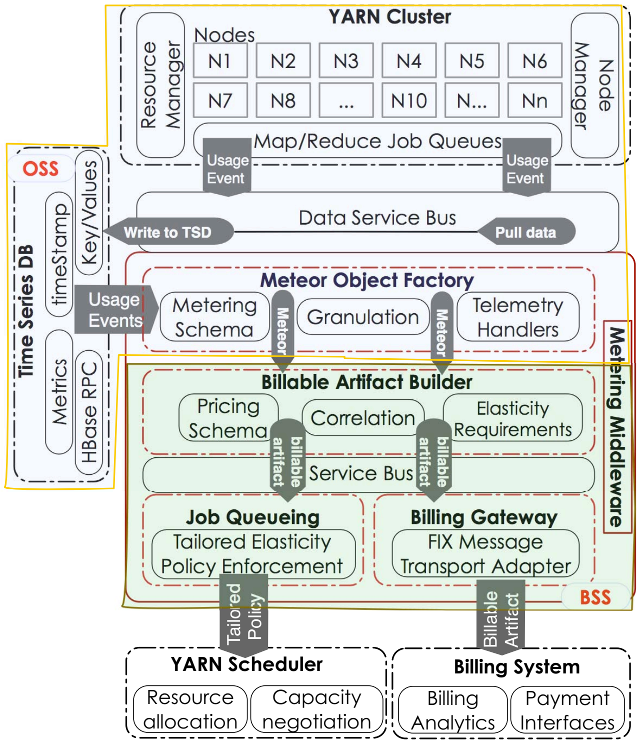

Fig. 2: MapReduce Job Metering & Rating Middleware Architecture.

The collection of monitored metrics data that are exposed by Hadoop Metrics2 system daemons are written into the time-series database to make it ready for querying and processing. We register our sinks to retrieve our desired metrics. The metering event data along with its job metadata is streamed to a tree-like[8] schema in a time series database to reflect our metering granularity in hierarchical layering. The metering tree schema is a JSON object that defines the hierarchical granularity topology in a tree model. TED traverses the tree via its HTTP API endpoint for further metering processes. The root(depth: 0) of the metering tree is the YARN cluster. One depth level below root is the DAGs-flow where the workflow of mapreduce jobs are hosted at the depth of 1. Moving forward, the MapReduce DAG branch is exposed at depth 2. Next, the depth increases to 3 where the MapReduce Job branch resides. This leads to a deeper branch of depth 4 where leaves exist with the map and reduce task's granularity. Navigating to the leaves of the tree represents the actual data points for metering events. With this approach implemented, TED will receive metering events and data for the completed tasks in the form of REST calls and JSON formatted data.

B. *Meteor Object Factory (MOF)*

So far, we have retrieved the usage data and loaded in the time series database in a dynamically built tree-like schema to keep the granularity. Once the metering data has been accumulated, the MOF component navigates the tree branches and their leaves as an input for constructing the *Meteor* objects. The MOF parses the metering events to extract resource usage `keys` and `values`. The meteor objects are basically aggregated metering events grouped by specific granularity and a user ID. For instance, a meteor on the MR-app granularity contains all the jobs' aggregated MR-tasks resource usage into a summarized JSON object. We call this object, a Meteor which will be processed and used to charge the user. Since the user already knows the size of their coming meteor, they then can be prepared in advance for that financially, etc. A meteor object structure is driven by `[key,value]` parity. The actual meteor that we generate carries 8 elements of which 6 represent the metrics.

To take a closer look of a meteor, think of metering the usage of a service. If service metering schema indicates that the metering pattern should be based on the number of service invocations. Then, having 10 calls from a service subscriber enables the telemetry instrumentation system to collect 10 metering events and the MOF generates 1 Meteor object indicating the service was invoked 10 times with the summary of resource usage. The process of meteor construction can be carried out on a predefined frequency of data collection. Meteors are built at various level of granularity that makes it easier to interpret, discover trends, gain insights into usage and performance for future elasticity strategy/policy selection and enforcement. When construction and transmission of some meteors might have priority in terms of their influence in elasticity decision making, then the priority queue pattern is considered in its life-cycle.

C. *Telemetry Handler*

The Telemetry Handler (TH) frequently invokes the *Granulation* REST API to generate the *Meteors* by aggregating the

[6]http://localhost:8080/jobstatus.php?jobId=$jobId&amp;jobStatus=$jobStatus
[7]http://www.w3.org/TR/xmlp-scenarios/#S1

[8]We use OpenTSDB 2.0 tree structure, a hierarchical method of organizing time-series into an easily navigable structure.

usage events into granules defined in the metering schema. Each meteor represents one completed job's aggregated resource usage. Then TH observes and audits the meteors JSON objects with the elasticity requirements to detect if any threshold is hit. The proper allocation policy (Scale Up/Down) will be plugged in if any check constraint is violated.

D. Billable Artifact Builder (BAB)

The constructed meteors in the MOF component will be transmitted to the BAB component. The BAB rates and monetizes the meteors by enriching them with the associated pricing schema and the user profile into a billable artifacts. Billable artifacts are granular resource usage-centric constructs capturing financial valuation of the abstract entities like job, map, reduce, etc. They indicate the econometric of the aggregate consumption data. The enriched meteors will be correlated with the elasticity controls for future policy enforcements, allocating or releasing resources, for instance. Job subscribers will be charged based on their usage pattern indicated in their billable artifacts. BAB enables an end-to-end mapping of the operational meteors and pricing schema to expose metered and billable artifacts to the billing system. This achieves a fine-grained unit of work for metering and pricing on the fly at economies of scale. BAB measures a number of metrics (CPU, storage, memory, etc) in `[key, value]` elements embraced by the granule and exposes them as billable artifacts to billing systems. Moreover, this exposure reveals the cost incurred on currently running mapreduce-based applications. The spending meter is updated at intervals to keep users aware of their payment stream in real time.

E. FIX Billing Gateway

Financial Information eXchange (FIX)[9] protocol is an open messaging specification to streamline electronic communications among financial entities for trade allocation, order submissions, etc. FIX billing gateway offers an architecture for exposing the billable artifacts to external billing systems (BS) and keep the pricing and metering schemata updated from partner endpoints.

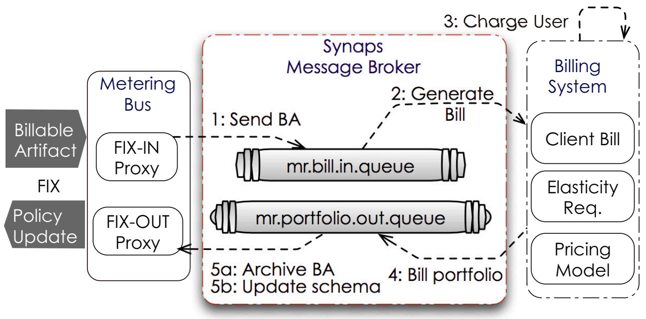

Fig. 3: TED tunnels billable artifacts using FIX protocol.

In our solution the metering service bus acts as the core message gateway, sending various billable artifacts and services to FIX endpoints using the built-in FIX transport adapter. The FIX message gateway connects endpoints by transforming messages to standard FIX messages using its base data dictionary and specifications. TED FIX protocol implementation is illustrated in figure 3. Proxy services are configured to transport billable artifacts in FIX messages to BS via message broker (*i.e.,* Apache Synapse[10]). The service bus converts FIX messages into XML which will be wrapped inside the Synapse message and sent to the BS. The FIX transport layer maintains session message correlation using message-id and correlation-id that allows the ESB to send relevant executions and acknowledgments back to the original FIX endpoint.

F. TED Job Queueing

Queueing theory is modelled on how to serve many arriving jobs while having scarce resources. In YARN, a queue is a logical collection of applications with a guaranteed resource capacity. They reflect the economies of resource allocation policies. Given the job's resource requirements, the goal is to improve performance by deploying a more optimized scheduling policy to achieve economies of scale. Our tailored policy in this study is to move the application to a queue which satisfies the application launch context. It is an indication that the following metrics underlying our model have been considered in rating the target queue for the proper positioning of the application.

◇ *Queue Utilization*: (ρ_i) is the fraction of time a container i is in use (non-idle). It is calculated by total observation of busy time (T_b) over length of observation period (τ), that can be formulated in $\rho_i = \frac{T_b}{\tau}$ equation.

◇ *Queue Throughput*: (Xi) is the rate of task completion (e.g., jobs/sec) at container i. Formally, the total number of completed jobs, J at container i within period of (τ) results in $Xi = \frac{J_{vm}}{\tau}$ throughput.

◇ *MapReduce Job Size*: (S_{mr}) indicates the amount of required time to run a job on the specific CPU alone. $(E[S_{mr}])$ represents the average required time to run the job excluding the queueing time (e.g., $\frac{1}{4}sec$).

◇ *Job Average Arrival Rate*: (λ) is the average rate of the job's arrival to the queue (e.g., λ = 2 jobs/s).

◇ *Job Average Serving Rate*: (μ) is the job's average serving rate in the queue (e.g., μ = 4 jobs/s = $\frac{1}{E[S_{mr}]}$).

◇ *Price of Entry & Cost of Waiting* : (P) is the job's entry price in the queue and the c indicates the cost per unit time of waiting in the queue (e.g., P = 2 € and c = 10 $cents/s$).

◇ *Job Migration Cost*: (M_c) contains the active job migration cost (e.g., M_c = 50 $cents$ per/job-size). This might come with a high cost because of state (memory). Such costs will be amortized over the new queue on its remaining processing time.

◇ *User Budget*: (B_u) contains the initial user budget limit to run the job.

In our model, after the migration, we enforce the shortest remaining processing time (SRPT) priority algorithm as a

[9]http://www.fixtradingcommunity.org/

[10]http://synapse.apache.org/

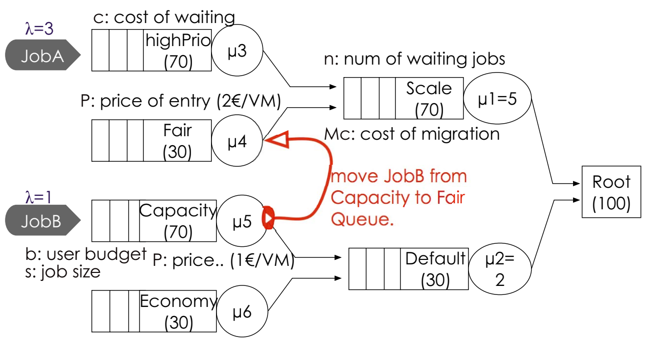

Fig. 4: TED Hierarchical Queueing Model & Job Migration.

serving mechanism for the migrated jobs. TED's queueing model is illustrated in Fig 4. In this mode, the used capacity of any parent queue is defined as the aggregate sum of used capacity of all the descendant queues recursively. the used capacity of a leaf queue is the amount of resources that are used by allocated containers of all applications running in that queue. Such a container is a unit of resource allocation across multiple resource types incorporating resource elements such as memory, CPU, disk, network etc, to execute a specific task of the application.

Algorithm 1 shows how applications are moved across queues in TED. The scheduling algorithm clarifies when to move which application to another queue. It is implemented and evaluated in the following section. The algorithm applies only for running jobs which are submitted on a specific queue.

Algorithm 1 TED scheduling algorithm for running jobs

```
procedure MoveToQueue($AppID, targetQueue$)
    if (AppStatus == ”Running”
    && AppQueue == ”Cloudera”){
    mapTasks[] ← list of maps of current Job
    reduceTasks[] ← list of reduces of current Job
    //check for the first completed map task
    for (each task in mapTasks)
    if (state of task equals ”Succeeded”)
    completedMapTaskId←currentMapTaskId;
    Compute $\rho_i$ and $Xi$ to classify highPriority queues
    if (completedMapTaskId ≠ null){
    // read the completed map task information from tsdb
    CPU ← store CPU usage of map task (ms);
    // Store HEAP & VMEM & PMEM values used by mapper
    costPerMap = CPU × targetQueue_cpuCost
    // plus the sum of HEAP, VMEM and PMEM;
    //estimate cost of job on the target higherPriority queue
    totalEstimatedCost=costPerMap×num.mapTasks + costPerMap×num.reduceTasks+P;
    if (totalEstimatedCost < user_budget)
    app.setQueue(targetQueue); }
    }
```

IV. Model Evaluation

Now, we present results from our real-world observations that show the efficiency of our model. We have implemented two threads; one for streaming the usage metrics data to tsdb and the other one for creating the meteors, cost estimations and migrating the job to the target queue. As a real job, we executed the YARN Pi example which computes the Pi value with the given precision with two configurations. In the first case, we have run the Pi job on 60 mappers with 30 samples per map. As for the environment set up, the Fair scheduler of YARN is in place and we have two queues of lowPriority ($weight = 1$) and highPririty ($weight = 2$). The observations on CPU consumption growth is trending over queue change in time. The aggregate results imply utility and are summarized in Fig 5.

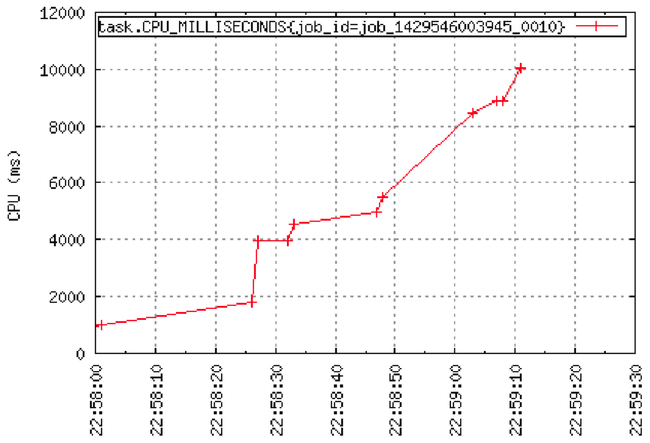

Fig. 5: YARN Pi example with 60 mappers and 30 samples (OpenTSDB *sum* diagram).

Taking these results together, four points stand out in this evaluation. First, the job was running in the first queue for about 25 seconds. Second, TED decides to migrate the job to a higherPriority queue after 25 seconds of jobs execution in which only 4 maps have been executed. Third, the migration takes 5 seconds and the job continues to run in the new queue at the starting time of 22 : 58 : 30. Finally, the first 4 maps were executed in 15 seconds while after migration the remaining 56 maps together with reduces executed in 40 seconds in the new queue.

Running the Pi with the second configuration results in Fig 6. It also took almost 4 maps to enforce the migration to the new queue at 02 : 33 : 00 which took 3 minutes to move to the new queue due to the larger size of the job. Having the migration in place, we can observe that the job is utilizing more resources to be finished. Since we had the SRPT policy in place, at the end of job, we see that it has the highest priority to be finished sooner. Results presented above are convincing enough to lead us to ascertain that in the highPriority queue the allocated memory and CPU were considerably higher to execute the job faster than the time it was submitted to lowPriority queue.

The empirical evidence observed in this study suggests that our consumption-based pricing model for MapReduce jobs provides statistically and economically important insights into financial behavior of an underlying resource usage model. TED enables applications to implement “elasticity-aware policies” to satisfy their resource requirements. If an associated

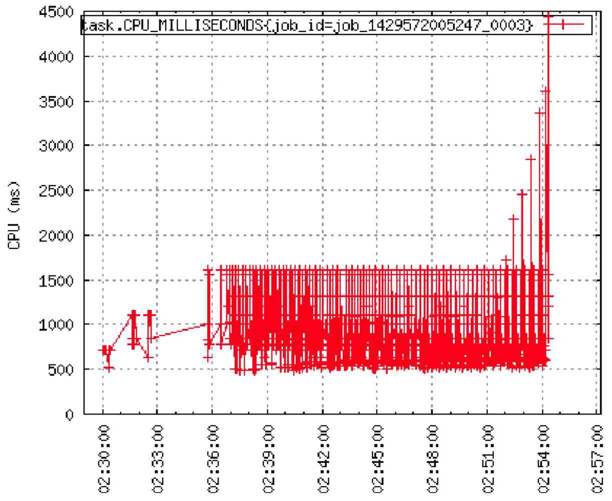

Fig. 6: YARN Pi example with 300 mappers and 100 samples (OpenTSDB *minmax* diagram).

job migration policy among queues is triggered, the allocation is leveled to the pre-configured queue capacity dynamically. This capacity leveling keeps the amount of resource allocation within the range (minSize and maxSize) of an intended resource allocation.

V. Related Work

To the best of our knowledge, this is the first paper that leverages the metering to the data processing domain. Meanwhile, there is some commendable research regarding the cloud service usage metering. Elmsroth et al.[6] proposed a loosely coupled architecture solution for an accounting and billing system for use in the RESERVOIR[7] project. In a separate work, Prasad et el.[8] described the usage of open source tools such as OpenTSDB, Hbase and Hadoop to store time series data and perform analytics on the time series data to get useful insights related to power consumption and get the result in pictorial format, essentially a graph. These papers do not provide any implementation or evaluation of the proposed architecture. There are some alternatives that propose billing and metering solutions, Narayan et al.[9]. Petersson[10] describes cloud metering and billing solution. Naik et al.[11] proposed a solution for metering of services delivered from multiple cloud providers. They incorporate the cloud service broker together with a metering control system to report metered data at configurable intervals. Their solution is developed and deployed as a plugin for IBM SmartCloud Enterprise. None of these solutions address mapreduce-based application metering and is not applicable to a cloud-based data processing framework like the Hadoop environment.

In relation to our approach, the SequenceIQ [11] project brings SLA policy based auto-scaling to Hadoop YARN. Their project is quite established and now is part of Hortonworks. It is built on top of YARN schedulers, allows to associate SLA policies to individual applications. It monitors application progress and apply scaling policies based on their CPU and memory usage. In contrast to all the noted related work, our TED middleware addresses granular metering of mapreduce jobs while considering the clients cost constraint regarding their jobs resource usage.

[11] http://sequenceiq.com

VI. Conclusion

We have presented an elastic data telemetry system that enables granular metering and automatic control of MapReduce applications due to their current behavior and preset configurations. TED is designed to enhance the resource utilization using YARN cluster queueing system. It observes the running job's progress, interprets its cost and quality behavior, audits the predefined job requirements. Then it generates a tailored resource allocation policy with regard to capacity constraints and moves the job to the potential queues to scale in or up.

So far, the authors offered and implemented a solution for the YARN environment which provides a data processing telemetry solution for mapreduce-based applications. As an outlook, our future work includes further extension to the TED model to formulate intuitions for best priority queue selection in multi-queue systems. We then move on to analyzing trees of queues with trendy metrics like *slowdown*, *starvation* and *fairness* for a workflow of jobs.

References

[1] M. Rappa, "The utility business model and the future of computing services," *IBM Systems Journal*, vol. 43, no. 1, pp. 32–42, 2004.

[2] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Gener. Comput. Syst.*, vol. 25, no. 6, pp. 599–616, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.future.2008.12.001

[3] S. Sen, C. Joe-Wong, S. Ha, and M. Chiang, "A survey of smart data pricing: Past proposals, current plans, and future trends," *ACM Comput. Surv.*, vol. 46, no. 2, pp. 15:1–15:37, Nov. 2013. [Online]. Available: http://doi.acm.org/10.1145/2543581.2543582

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[5] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong, "Principles of elastic processes," *IEEE Internet Computing*, vol. 15, no. 5, pp. 66–71, 2011.

[6] E. Elmroth, F. G. Marquez, D. Henriksson, and D. P. Ferrera, "Accounting and billing for federated cloud infrastructures," in *Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing*, ser. GCC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 268–275. [Online]. Available: http://dx.doi.org/10.1109/GCC.2009.37

[7] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Cáceres, M. Ben-Yehuda, W. Emmerich, and F. Galán, "The reservoir model and architecture for open federated cloud computing," *IBM J. Res. Dev.*, vol. 53, no. 4, pp. 535–545, Jul. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1850659.1850663

[8] S. Prasad and S. Avinash, "Smart meter data analytics using opentsdb and hadoop," in *Innovative Smart Grid Technologies - Asia (ISGT Asia), 2013 IEEE*, Nov 2013, pp. 1–6.

[9] A. Narayan, S. Rao, G. Ranjan, and K. Dheenadayalan, "Smart metering of cloud services," in *Systems Conference (SysCon), 2012 IEEE International*, March 2012, pp. 1–7.

[10] J. Petersson, "Cloud metering and billing," http://www.ibm.com/developerworks/cloud/library/cl-cloudmetering [Online; accessed 08-August-2011].

[11] V. Naik, K. Beaty, and A. Kundu, "Service usage metering in hybrid cloud environments," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, March 2014, pp. 253–260.