RI 05002 July 1, 2004 Computer Science

IBM Research Report

Metric Service : A Mediation Engine for Metrics Collection, Aggregation and Composition

Arun Kumar, Vikas Agarwal

IBM Research Division
IBM India Research Lab
Block I, I.I.T. Campus, Hauz Khas
New Delhi - 110016. India.

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

Metric Service : A Mediation Engine for Metrics Collection, Aggregation and Composition

Arun Kumar, Vikas Agarwal
{kkarun, avikas}@in.ibm.com
IBM India Research Laboratory, New Delhi, INDIA

Abstract

Various instrumented systems produce metrics that capture the state of the system at different points of time. These could be in the form of logs written to disk or data records made available to interested consumers through messaging systems. Management applications utilize such metrics to derive information needed for decision making. Such applications include accounting, fault handling, intrusion detection, resource provisioning etc. However, each such application is typically custom built to encode the data manipulation logic specific to the management task at hand. These custom solutions are non-reusable, non-shareable and become increasingly complex and hard to manage as they evolve to meet ever changing needs of the environment. We present Metric Service, a mediation engine that can be shared and configured to perform data manipulation operations for a wide variety of management applications. It allows application specific manipulation of raw metrics to be performed in the middleware itself and also enables integration of data from heterogeneous systems that may be geographically distributed . We also introduce MS-Policy, an XML based policy specification language that forms the core of the proposed engine. It allows the data integration and manipulation logic, for a particular management task, to be specified in terms of metrics collection, aggregation and composition. We describe our prototype implementation and report the results of performance experiments.

Keywords: XML, Grid Services, Metrics, Mediation, Aggregation

1 Introduction

Most software systems generate monitoring data either reported as various kinds of logs or made available through messaging systems. This data typically contains metrics that capture the state information of the associated system at various points in time. Such information has immense value from management point of view as it is used for decision making purposes. It could be used to study system behavior, to detect performance bottlenecks, to recover from faults, for enforcing service level agreements and so on.

However, most often there is a gap between the information that a management application requires and what is made available by existing instrumentation or monitoring agents. This is because monitoring usually comes with a price in terms of performance degradation. System designers are forced to be selective about the information that they can make available. Only those metrics are computed that are deemed important by them. The reporting frequency and the data format used for making the metrics available is also geared towards minimizing the performance hit. Therefore, management applications need to derive desired information, in terms of computing new metrics, by applying some manipulation process to the metrics that are available through existing instrumentation. Furthermore, many a times it is necessary to be able to collect and aggregate information from multiple data sources. This is non-trivial especially since the monitored data may be available from heterogeneous and geographically distributed systems.

Mediator architecture [19] was proposed for such scenarios. A software module, called *mediator*, exploits encoded knowledge about the available data, to create information needed by a higher layer of applications. It is accompanied by *wrappers* that transform information available from data sources into a common model understood by the mediator. However, most systems [10, 15, 20] utilize the mediation concept only for solving problems such as information integration from heterogeneous data sources and data conversion from one format to another.

We present *Metric Service* (refer Fig. 1) – a mediation engine that goes beyond addressing the data

integration and conversion problem. It provides a mechanism for declaratively specifying and executing the logic for creation of desired information required for decision making. Such capability enables entire management tasks, that are usually custom-built every time, to be made available in the middleware itself. It is similar to publish-subscribe systems [17] in the sense that it receives a stream of raw metrics from a number of data sources (*producers*) and transforms them into composite metrics desired by management applications (*consumers*). However, the transformation process can be specified declaratively and does more than just filtering. It involves *aggregation*, i.e. computing aggregate values from multiple instances of a single metric and/or *composition*, i.e. computing the value of a metric from instances of two or more different metrics. The values of metric(s) being aggregated or composed may come from multiple sources and their arrival may be offset in time.

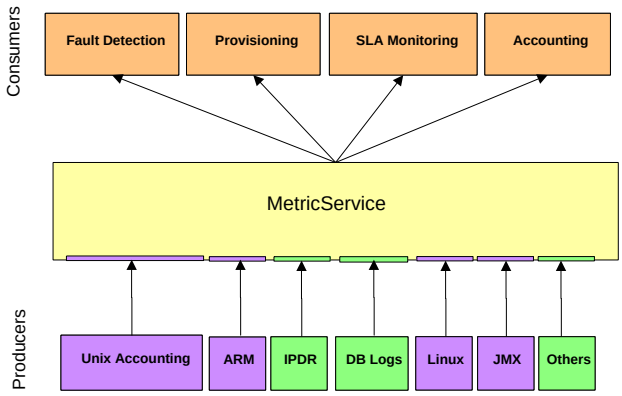

Figure 1: Metric Service Block Diagram

At the core of the Metric Service is *MS-Policy* – an XML based language that allows the specification of the metrics manipulation process in terms of metrics collection, aggregation and composition. Each such specification becomes the definition of a mediator and is executed concurrently with others in the Metric Service. The engine can be embedded inside other systems or it could be made available as a stand-alone web service [6] or grid service [5].

1.1 An Example Scenario - Accounting

To illustrate the working of Metric Service, we chose the example of accounting on UNIX servers as the management task. Most UNIX based operating systems provide accounting capabilities in terms of process accounting, project accounting, system activity reporting etc. [7]. The operating system modules that monitor various processes and periodically update system logs are the producers of metrics, in this scenario. The consumers of such information include management applications such as resource provisioning, cluster management systems, fault handling applications etc.

Specifically, *process accounting* provides a capability in the O/S to record a collection of information for each and every process completed. The log file created is called *pacct* and each log entry includes metrics such as user-id, group-id, cpu usage, memory usage etc. Connection related metrics such as login, logout, system reboot time etc. are logged in *wtmp* log. Similarly, disk usage and printer usage records are logged in *dtmp* and *qacct* respectively.

As shown in Fig. 2, the *runacct* module of the accounting system, implements a management task that executes daily to process and report accounting data. It collects the information from various O/S accounting logs such as *pacct*, *dtmp*, *wtmp* etc. and summarizes them into total accounting records (*tacct*s). This summarization is done on per user basis and involves metric aggregation, metric composition and event correlation. Examples of these include computing daily totals of some metrics such as *bytesTransferredForIO*, computing *cpuUsage* from two metrics *systemCpuTime* and *userCpuTime* and computing *connectionTime* by correlating *login* and *logout* records respectively.

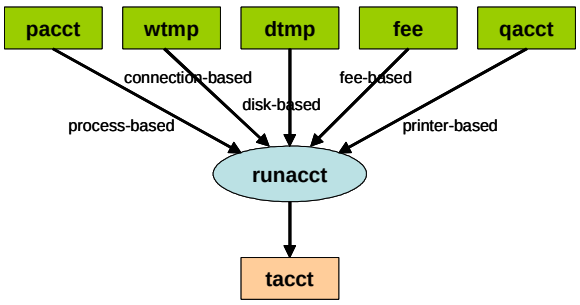

Figure 2: Unix Process Accounting

The management software that implements *runacct* functionality is currently custom-developed for each O/S platform. With Metric Service middleware available, it can be implemented on a given O/S platform by simply specifying the functionality for that platform in MS-Policy language and instantiating it in a Metric Service instance. In this scenario, the above approach, not only helps in managing heterogeneity and promoting software reuse but enables other types of accounting as well. For instance, and as described

later, it can be used to provide unified accounting for a cluster of heterogeneous servers using existing log mechanisms.

The rest of the paper is organized as follows. Section 2 describes the MS-Policy language. Design and implementation of the Metric Service engine is discussed in Sec. 3. Section 4 presents the results of our performance evaluation experiments. Section 5 discusses some scalability issues and applications of Metric Service. Finally, Sec. 6 discusses related work and we conclude in Sec. 7.

2 MS-Policy

Metric Service, being a middleware component, has to be flexible so that the metrics manipulation logic specific to each management task can be configured into it easily. MS-Policy serves as a language for specifying that logic whereas Metric Service instantiates that logic as a mediator and executes it. Several issues related to the main aspects – collection, aggregation and composition of metrics – influence the design of MS-Policy language. This section identifies those issues and describes important elements of the MS-Policy language [9].

2.1 Representation Language

We use XML Schema Definition Language (XSD)[1] as the meta language in which the structure and basic constructs of the MS-Policy language are defined. Each policy instance is, therefore, an XML document conforming to this schema. Traditionally, XML has been used to define, share, and integrate information from heterogeneous systems and applications. Here, we go beyond that and apply its structured nature to enable computation logic to be expressed in an XML based language. Extensibility of the MS-Policy language comes as an added benefit by virtue of extensible nature of XML.

Since Metric Service collects metrics from heterogeneous systems, it also has to deal with the traditional problems of information integration and data conversion. Mediator architecture comes to rescue here. A *wrapper* module obtains raw metrics from the data producers (i.e. log entries, instrumented applications etc.) and converts them into a format understood by the Metric Service engine. We rely on XML again and use it to specify *metric record* as that common format into which all the disparate log entries get transformed. A metric record is defined as a collection of individual reported metrics, some of which may represent context information such as timestamp, user-id, machine-id etc. We consciously chose to represent metrics in a record because we observed that each metric is meaningful only in the scope of its context and a context is common to many metrics. Moreover, we distinguish between a *metric record definition* and a *metric record instance* which is a good design for extensibility as demonstrated in [3, 12].

A *metric record definition* allows specification of different types of metric records each of which is given a unique name. A policy can then refer to these metric record definitions to specify operations on their instances. Data handling capability of Metric Service is, therefore, extensible since adding support for new data simply amounts to adding new metric record definitions and referring to them in the policies. Figure 3 shows a fragment of *pacct* metric record definition and Fig. 4 shows an example *pacct* record instance as sent by the wrappers.

```
<MetricRecordDef name="http://www.ibm.com/ogsa/schema/MetricService/pacct">
  <MetricDef>
    <!-- user id-->
    <Name>ac_uid</Name>
    <DataType>integer</DataType>
  </MetricDef>
  <MetricDef>
    <!-- group id -->
    <Name>ac_gid</Name>
     <DataType>integer</DataType>
  </MetricDef>
  <MetricDef>
    <!-- begin time -->
    <Name>ac_btime</Name>
     <DataType>dateTime</DataType>
  </MetricDef>
   .....
   .....
</MetricRecordDef>
```

Figure 3: Metric Record Definition for pacct

```
<MetricRecord metricRecordDefName="http://www.ibm.com/ogsa/schema/
                                MetricService/pacct">
    <Metric name="ac_uid">123</Metric>
    <Metric name="ac_gid">123</Metric>
    <Metric name="ac_btime">2003-11-06T14:20:34Z</Metric>
    <Metric name="ac_utime">0.238 </Metric>
    <Metric name="ac_stime">0.193</Metric>
    <Metric name="ac_etime">0.527</Metric>
    <Metric name="ac_mem">12.35</Metric>
    <Metric name="ac_io">58</Metric>
    <Metric name="ac_rw">0</Metric>
    <Metric name="ac_comm">ps</Metric>
</MetricRecord>
```

Figure 4: Metric Record instance for pacct

2.2 Policy Structure

We now describe the structure of a policy specified using MS-Policy language. It has four components encapsulated in a *MetricRecordDirectiveType* (refer

[1] http://www.w3.org/XML/Schema

Fig. 5). The core of the policy, i.e. the metrics manipulation process, is captured in *MetricDirective* elements. These are supported by:

- *TargetMetricRecord* elements that specify *which* metric records have to be computed,
- *Triggers* that specify *when* those records or their intermediate metrics have to be computed, and
- *InputMetricRecord* elements that specify the input metrics, their source and the mechanism to obtain them.

To enable Metric Service to be shareable across management applications, it is important to allow simultaneous execution of multiple concurrent instances of such policies. Therefore, each policy is identified by a name and is instantiated with its own set of target metric record instances. To apply the same metric manipulation process for different sets of producers/consumers, a copy of the original policy is made by the administrator and the source of raw input metrics is modified in its *InputMetricRecord* elements. This modified policy is then instantiated with a different name.

Support for multiple concurrent instances also places other design requirements. The Metric Service has to take into account the fact that an incoming metric record may be consumed by more than one policy instance. On the other hand, instances of same kind of metric record but arriving from different sources may have to be fed to different policies. It has to take care that an event (identified by a trigger) may have to be delivered to different policy instances and multiple fragments of each policy. It also has to distinguish between intermediate and final metric instances getting computed for different policy instances. The policy constructs have been designed to support these situations.

2.2.1 TargetMetricRecord

A TargetMetricRecord element specifies characteristics of a metric record that would contain, at the end of policy execution, metrics desired by that policy's consumer(s). The type of the record represented by a target metric record is identified by a *MetricRecordDefName* element. Since multiple policies may share *metric record definition*s, the definitions for *tacct* and all other metric record types are configured into Metric Service engine separately. Management applications may wish to obtain derived metrics computed for multiple groups of managed elements. For instance, accounting may be done on a per user basis or per project basis, resource provisioning may

```
<complexType name="MetricRecordDirectiveType">
    <sequence>
        <element name="TargetMetricRecord" type="mrd:TargetMetricRecordType"
            minOccurs="1" maxOccurs="unbounded">
        </element>

        <element name="Trigger"  type="mrd:TriggerType"
            minOccurs="0" maxOccurs="unbounded">
        </element>

        <element name="InputMetricRecord" type="mrd:InputMetricRecordType"
            minOccurs="1" maxOccurs="unbounded">
        </element>

        <element name="MetricDirective" type="mrd:MetricDirectiveType"
            minOccurs="1" maxOccurs="unbounded">
        </element>
    </sequence>
    <attribute name="name" type="string" />
</complexType>
```

Figure 5: Metric Record Directive (MS-Policy)

be done differently for different classes of customers, and so on. To cater to this requirement, MS-Policy supports the notion of a *key* that consists of a set of one or more metrics of the input metric records. It is represented by *KeyMetric* elements. Derived metrics are grouped on the basis of distinct values of the KeyMetric elements and are collected into separate target metric record instances – one for each group. The combination of the values of MetricRecordDefName and KeyMetric elements uniquely identifies a target metric record within the scope of a policy.

A *TriggerChoice* element captures the time intervals or the event on the occurrence of which the target metric record should be computed. It represents a choice between a new *Trigger* (described next) or a reference to an existing one defined somewhere else in the policy. A *MetricRecordPersistence* element specifies the retention policy for this target metric record. Figure 6 shows a policy fragment for specifying a target metric record of type *tacct*.

```
<TargetMetricRecord>
    <MetricRecordDefName>
        http://www.ibm.com/ogsa/schema/MetricService/tacct
    </MetricRecordDefName>
    <KeyMetric metricName="ta_uid">
        <InputRecordKey metricRecordDefName="http://www.ibm.com/ogsa/
                                schema/MetricService/pacct">
            ac_uid
        </InputRecordKey>
    </KeyMetric>
    <TriggerChoice>
        <TriggerName>RD1</TriggerName>
    </TriggerChoice>
    <MetricRecordPersistenceDirective>
        <StorageDuration>P6Y</StorageDuration>
    </MetricRecordPersistenceDirective>
</TargetMetricRecord>
```

Figure 6: Target Metric Record

```
<Trigger name="RD1" xsi:type="ScheduleType">
    <startDate>2003-11-21T11:00:00Z</startDate>
    <endDate>2006-11-21T14:25:00Z</endDate>
    <interval>PT30S</interval>
</Trigger>

<Trigger name="RD2" xsi:type="LogicalExpressionType">
    <Predicate operator="Equal">
        <Operand>
            <MetricOperand>
                <MetricDirectiveName>
                    ut_typeDirective
                </MetricDirectiveName>
            </MetricOperand>
        </Operand>
        <Operand>
            <Constant dataType="string">RUN_LEVEL</Constant>
        </Operand>
    </Predicate>
</Trigger>
```

Figure 7: A Schedule and a LogicalExpression

2.2.2 Triggers

Triggers define the event model of MS-Policy. Each trigger is activated either by the occurrence of an event or at pre-determined points in time. On activation, a trigger initiates the action or a list of actions associated with it. Trigger type is abstract and is extended into concrete elements *LogicalExpression* and *Schedule*. A Schedule is specified in terms of start time, end time and an interval at which the schedule is triggered. It could be used for purposes such as periodically launching an action for pulling input metrics from their sources or for driving construction of a time-series to be used for computing some statistical aggregation function.

A LogicalExpression represents a condition which, when evaluated to true, results in the execution of an action. It is modeled as a boolean expression with relational operators applied to input or derived metrics. The evaluation of a logical expression happens when the values of all the involved metrics become available. It could be used in situations where an action needs to be triggered on the occurrence of an event such as detection and handling of faults, provisioning of resources under heavy load etc. Figure 7 shows an example of each type.

All common triggers of a policy are placed in the Trigger element of MetricRecordDirective (refer Fig. 5) and different portions of a policy either use them or define their own. Use of common triggers achieves the effect of synchronization among those policy portions. Triggers, as specified above, are local to a policy but the Metric Service engine can optimize by instantiating a single instance if a trigger is common across policy instances.

```
<MetricDirective name="ta_rwDirective"
                xsi:type="CompositeMetricDirectiveType" initialValue="0.0">
    <MetricName metricRecordDefName="http://www.ibm.com/ogsa/
                schema/MetricService/tacct">
        ta_rw
    </MetricName>
    <Function resultType="double" xsi:type="SumAccumulatorFunction"
                                                    initialValue="0">
        <Operand>
            <MetricOperand>
                <MetricDirectiveName>ac_rwDirective</MetricDirectiveName>
            </MetricOperand>
        </Operand>
        <TriggerChoice>
            <TriggerName>RD1</TriggerName>
        </TriggerChoice>
    </Function>
</MetricDirective>

<MetricDirective name="ac_rwDirective" xsi:type="LeafMetricDirectiveType">
    <MetricName metricRecordDefName="http://www.ibm.com/ogsa/
                schema/MetricService/pacct">
        ac_rw
    </MetricName>
    <InputMetricRecordName>
        http://www.ibm.com/irl/OSlogs/pacct_push
    </InputMetricRecordName>
</MetricDirective>
```

Figure 8: Metric Directives

2.2.3 MetricDirectives

A MetricDirective element specifies how a metric's value is obtained. It contains an element *MetricName* that specifies the name of the metric for which the metric directive is defined. Collection of MetricDirectives of a policy represents the metric manipulation process enabled by that policy.

MetricDirective is an abstract type and is extended into a *LeafMetricDirective* and a *CompositeMetricDirective*. LeafMetricDirective is used when the desired metric is directly available as a raw input metric. It simply refers to an *InputMetricRecord* (described next) that contains the desired metric. For other metrics that are derived from raw metrics, *CompositeMetricDirective* specifies the aggregation and composition process. It consists of an element *Function* and an attribute *initialValue*. The attribute *initialValue* allows a default value to be used, wherever acceptable, if no new value could be computed for the metric on the expiry of a schedule. The element Function is also an abstract type and acts as a placeholder for various mathematical functions such as plus, divide etc., and statistical functions such as mean, max, percentile etc. that have been modeled as its concrete subtypes. Functions operate on other functions, available metrics (specified by their metric directives) and constants. Functions that act on multiple values of the same metric (e.g. sum, mean etc.) enable aggregation functionality. Other functions that operate on multiple metrics (e.g. divide, multiply etc.)

enable composition functionality. When completely specified, a function takes the form of a hierarchy of computations with input metrics as the leaves and a derived metric as the root.

There is a LeafMetricDirective for each input metric and a CompositeMetricDirective for each metric in the target metric record(s) as well as for all those intermediate metrics that may be shared among policy fragments. Sharing of a metric computation prevents duplication when multiple target metrics need to be derived using some common intermediate metrics. To enable this, each MetricDirective is assigned a unique name specified in its *name* attribute so that other metric directives can refer to it. The name of the MetricDirective is also used to register it as a receiver of new values of its operand metrics. Figure 8 shows an example composite metric directive for *ta_rw* metric in *tacct* record. *ta_rw* represents number of cumulative blocks read/written and is computed from *ac_rw* metric of *pacct* as represented by the leaf metric directive, also shown in the same figure.

2.2.4 InputMetricRecord

InputMetricRecord element is identified by a *name* attribute. It is used in a LeafMetricDirective to capture the details of the type of metric record whose instances would be accepted by the executing policy in order to obtain the required input metric(s).

It has a *MetricRecordDefName* element that refers to the definition of the required input metric record. The Metric Service uses the metric record definition names to select the appropriate input metric record instances, for each executing policy, from the set of all instances that it receives/obtains. The metric record definition represented by this name is also required to interpret the contents of the received metric record instances. A *Locator* element specifies the data-source(s) of this policy, i.e. the set of URLs from which the policy would accept metric record instances.

It also has a *CommunicationMethod* element that specifies the mechanism to be used for obtaining the input records. Depending upon the nature of wrappers available, the records may either be *push*ed to the Metric Service asynchronously or they may have to be *pull*ed synchronously from the wrappers. Both these semantics are supported in order to maintain the generality of the Metric Service. An optional *TriggerChoice* element specifies the schedule for pulling the input records from their sources. Figure 9 shows an example input metric record for *pacct* records.

```
<InputMetricRecord name="http://www.ibm.com/irl/OSlogs/pacct_push">
   <MetricRecordDefName>
      http://www.ibm.com/ogsa/schema/MetricService/pacct
   </MetricRecordDefName>
   <Locator>*.in.ibm.com</Locator>
   <CommunicationMethod>Push</CommunicationMethod>
</InputMetricRecord>
```

Figure 9: Input Metric Record

2.3 Accumulator

Metric aggregation functionality requires that a set of values of a metric be collected over an interval of time and then aggregated by applying an appropriate function such as sum, mean etc. In [8], Time-series and Queue data-structures have been used to accumulate these raw values periodically and asynchronously respectively. The function is then applied over these accumulated values. In addition to supporting those data-structures, we introduce the notion of an *accumulator* function that enables this functionality without having to store all the raw values, wherever possible. It essentially maintains a running aggregation of the raw values. A separate accumulator is defined to support each statistical function that can be computed in this manner. An accumulator has an associated trigger that determines when the accumulated value is ready to be reported. SumAccumulatorFunction has been used in the example in Fig. 8.

2.4 Event Correlation

While aggregating metrics, certain scenarios require that the incoming metric values be correlated across time. For instance, in the UNIX accounting example the *wtmp* record from the connection based accounting logs contains login and logout events for each user session. There may be system reboot events that may mark end of all open connections. For accounting purposes such events need to be detected and correlated to compute the effective user connection time represented by *ta_con* metric in *tacct* record. To support event correlation, there are different mechanisms available such as finite state machines, rule based reasoning, model based reasoning etc. [11].

We modeled a *deterministic finite state machine* in XML for handling such event correlation and incorporated it as a construct available in MS-Policy language. It can be used as an operand to a function. We specify *StateMachineType* as a collection of one or more *State* elements. Each State element is identified by an *id* attribute that has a value of *StartState* and *EndState* for the initial state and the terminating states respectively. A State consists of

a collection of *EventAction* elements each of which specifies an event accepted in that state and a set of actions to be executed on the occurrence of that event. EventAction element also contains a default action, *GotoState*, that is executed after all the other actions for that event have been executed. An event is represented by a *UserEvent* element that is of type Trigger. An action is represented by an *Action* element that contains a Function and an *isResult* attribute.

```
<StateMachine>
 <State id="StartState">
  <EventAction>
   <UserEvent>
    <Trigger>
     <Condition>
      <Predicate operator="Equal">
       <Operand>
        <MetricOperand>
         <MetricDirectiveName>ut_typeDirective</MetricDirectiveName>
        </MetricOperand>
       </Operand>
       <Operand>
        <Constant dataType="string">BOOT_TIME</Constant>
       </Operand>
      </Predicate>
     </Condition>
    </Trigger>
   </UserEvent>
   <Action>
    <Function resultType="boolean" xsi:type="StoreFunction">.....</Function>
   </Action>
   <Action>.....</Action>
   <GotoState>1</GotoState>
  </EventAction>
  .....
 </State>
 <State id="1">
  <EventAction>
   <UserEvent>.....</UserEvent>
   <Action isResult="true">.....</Action>
   <GotoState>EndState</GotoState>
  </EventAction>
  .....
 </State>
 .....
</StateMachine>
```

Figure 10: State Machine for Event Correlation

Our main purpose of modeling state machines is to correlate events and perform some computation on the basis of that correlation. The *isResult* attribute specifies whether the result of the associated action is to be treated as the final result of the computation performed by the state machine. It distinguishes the result from other intermediate computations and also specifies clearly what to propagate out once the state machine reaches the end state. The collection of all accepted events for all states forms the *input alphabet* of the state machine. Set of all <(State), (set of EventAction elements)> pairs captures its *transition function*.

Since we support multiple concurrent policies and multiple key values in each policy, event correlation mechanism has to distinguish between events arriving for different policies and within that for different entities (i.e. key values). Therefore, a separate state machine instance is created for each set of Key-Metric values in the target metric record. Each executing policy has its own state machine instances all of which are instantiated inside a *StateMachineContainer*. From the input metric records arriving at the Metric Service, the state machine container identifies the relevant events (i.e. metric-value pairs) for each state machine instance and delivers them. It also handles the creation of new state machine instances whenever an event for a new *key* value arrives. Apart from that it also propagates the results of computations to the Function element of which the state machine is an operand. Figure 10 shows a partial fragment of a state machine. A value of BOOT_TIME for the metric *utType* (available from *wtmp* logs and represented by *utType_directive*) is accepted as a valid event by its *StartState*. State with id 1 has an action that leads to *EndState*. Two special functions *Load* and *Store* are defined for state machines to provide internal data persistence across state transitions.

3 Design and Implementation

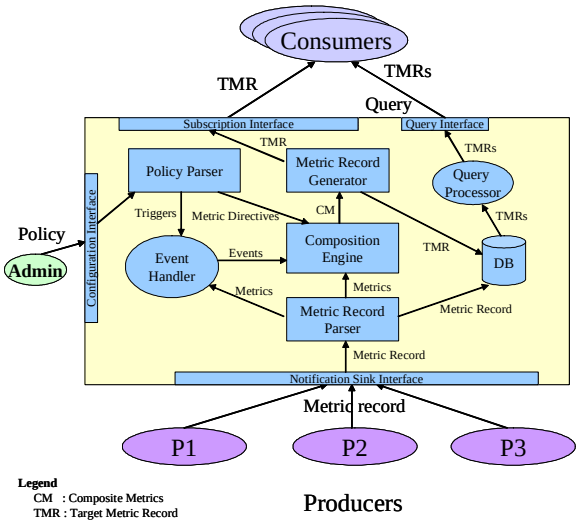

Figure 11: Metric Service Design

Figure 11 shows the software architecture of the Metric Service engine. Metric producers P1, P2, P3 are basically wrappers that retrieve or receive log entries from instrumented systems, convert them into appropriate metric records and send those metric records to the Metric Service. Consumers, on the other hand, represent management applications that are interested in one or more derived metrics. They may either subscribe to receive those metrics asynchronously

as and when they are computed or they may query the Metric Service, whenever required, to retrieve the composite metrics that have been computed. The Metric Service is given one or more policies specifying the mediators to be instantiated. We restrict this operation to an administrator even though the configuration interface of the Metric Service makes it possible for customers to add such policies directly. Since multiple active mediators can co-exist simultaneously, the Metric Service acts as a hosting service for them. A mediator once instantiated, executes until it is explicitly removed by the administrator or its target metric record's schedule expires.

3.1 Software Architecture

The Metric Service consists of the following components:

1. **Policy Parser** receives the policies from the *configuration interface*. It parses each policy and forwards its components to different modules. For instance, metric directives are sent to the *composition engine* and triggers to the *event handler*.

2. **Event Handler** allows triggers to be registered by the policy parser. If the trigger is a schedule, the event handler initiates the associated action at the intervals specified. For logical expressions, it evaluates the condition whenever new values of its operand metrics arrive and initiates the associated action if it evaluates to true.

3. **Notification Sink** module is responsible for receiving notifications, that contain metric records, from the producers (i.e. wrappers).

4. **Metric Record Parser** receives metric records from the notification sink interface, parses them and extracts the raw input metrics out of those. These are then forwarded to the entities that require them. Event handler uses some of these metrics to evaluate the triggers registered with it, the *composition engine* uses them in the derivation process of composite metrics.

5. **Composition Engine** is the heart of the Metric Service. It instantiates and executes the metric directives of the policies supplied to it by the policy parser. The triggers, of those policies, registered with the event handler control the behavior of the engine while computing composite metrics. Owing to the object oriented design of the composition engine and presence of context information in metric records, the Metric Service can execute multiple instances of the same policy or instances of different policies simultaneously.

6. **Target Metric Record Generator** receives individual computed metrics from the Composition Engine, and generates *target metric records* from those. These target metric records are then dispatched to the consumers who have subscribed to them, through the Subscription interface. These are also stored into the local database for later retrieval by the consumers.

7. **DB** is the local database that stores the input metric records as well as computed target metric records as per their persistence policies.

8. **Subscription** module allows consumers to subscribe to the target metric records that contain composite metrics of interest to them.

9. **Query** module is composed of a *query interface* that enables the consumers to submit queries and a *query processor* that evaluates these queries to retrieve stored target metric records that have been queried for.

3.2 Engine Implementation

We implemented the Metric Service Engine in Java[2] and provided it a grid service interface using Globus OGSA v3.0[3]. Each of the MS-Policy elements is implemented as a Java class with sub-elements implemented as components of the parent class. As mentioned earlier, due to the nature of Functions a CompositeMetricDirective takes the form of a hierarchy of computations. A separate instance of this hierarchy is created for each distinct value of the KeyMetric elements of the target metric record. The event model of the policy drives the computations in this hierarchy. In the current implementation, the metrics obtained from the input metric records enter through LeafMetricDirectives and are pushed up in the MetricDirective hierarchy. At each level the function(s) accept outputs of functions from lower levels, other metrics and constants as operands. The computation at this level leads to a new value that is pushed up to be used in further computations. The computation for a target metric stops when the value reaches the top element of the hierarchy.

[2] Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the U.S., other countries, or both.

[3] Globus Toolkit 3.0. http://www-unix.globus.org/toolkit

In order to support good request rate, the delivery of input metric records, to the policy elements that require them, should be optimized. This is non-trivial since an input metric record may be required by many elements across many executing policies. Moreover, the specific instances that get delivered to a policy depends upon the source of that metric record instance. We implemented a data-structure that, on arrival of first metric record instance from a particular source, creates and stores a mapping from the <data-source-address, metric-record-name> to the set of destination elements. For all subsequent instances of those input records, it takes $O(1)$ time to determine the set of destination elements where the metric record instance has to be delivered. Since actual delivery of metric records to the destination elements involves a synchronous call, we used a multi-threaded approach to avoid any bottlenecks in that phase. For delivery of composite metrics (in the form of target metric records), our current implementation supports subscription interface and we plan to support the query interface in future.

4 Performance Evaluation

We now present the details of the experiments that we conducted to study the performance of our Metric Service implementation. For the experiments, we required a client that could generate requests at specified rate for a given duration. The QoS parameters to be reported included *Achieved Request Rate*, *Achieved Reply Rate* and *Average Response Time*, among others, for each execution of the client. Since, we could not find a suitable client for grid services, we implemented our own multi-threaded client in Java similar to httperf [13].

4.1 Experimental Setup

The testbed consists of one server and one client system each running RedHat Linux on a 2.66 GHz Pentium IV processor[4], 2 GB RAM, and a 100 Mbps ethernet network interface. The server system runs Globus OGSA toolkit v3.0 on Apache Tomcat. Tests were conducted to study the following behaviors with increasing request rate:

1. **Effect of the policy complexity:** The complexity of a policy varies with the number and kind of computations involved in the metric derivation procedure. We studied the effect of increasing number of composite metrics to be computed on the server performance. We used *kcore* metric of *tacct* record, that represents cumulative memory usage, as our test metric. Each composite metric derivation process involved one division, one multiplication and one addition for each input metric record.

2. **Effect of number of policies:** Since multiple policies may exist and execute simultaneously in the Metric Service engine, we studied the effect of varying number of policies on the performance of the system. Each policy consisted of a single composite metric computation.

3. **Effect of number of Key values:** As explained in Sec. 2, for a given data source a separate target metric record is computed for each distinct value of KeyMetrics. The number of distinct values could be very large. For instance, a server in the Unix accounting example may have thousands of users. If *used-id* is the KeyMetric then on receipt of the first *pacct* record for a user, the Metric Service instantiates a separate MetricDirective instance corresponding to that user. We conducted experiments to study the behavior of the engine with respect to varying number of key metric values in the input data set.

In each of the above cases, to measure the maximum request rate that the server can handle before reaching saturation, the client sends requests at rate starting from 100 per sec. till the server saturates. Each request used a separate connection. Each experiment consisted of client sending requests at the specified rate for a duration of 120 seconds.

4.2 Results

Twenty runs of the above described experiments, for each request rate, were conducted. The results were measured as the observed reply rate i.e. total number of requests served divided by the experiment duration. Figures 12 to 14 depict the mean of the observed reply rate for twenty runs of various experiments.

Figure 12 shows the plot of the observed reply rate with the request rate for varying policy complexity. We performed this test for policies containing 1, 50, 100, 150, 200, and 250 composite metrics to be computed. From the graph we can observe that for the 1 metric case, the reply rate is same as the request rate till about 200 requests/sec. As the load is increased beyond this, the server begins to saturate and the

[4]Pentium is a trademark of Intel Corporation in the U.S., other countries, or both

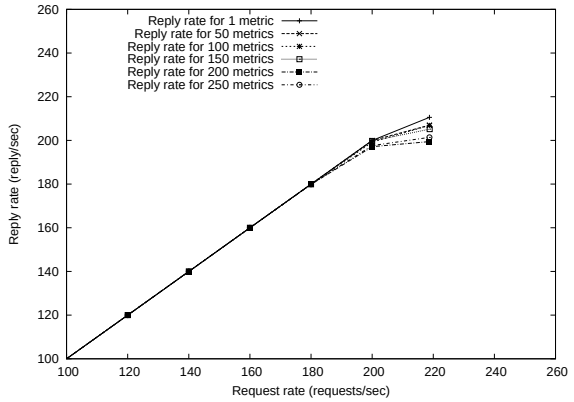

Figure 12: Effect of policy complexity

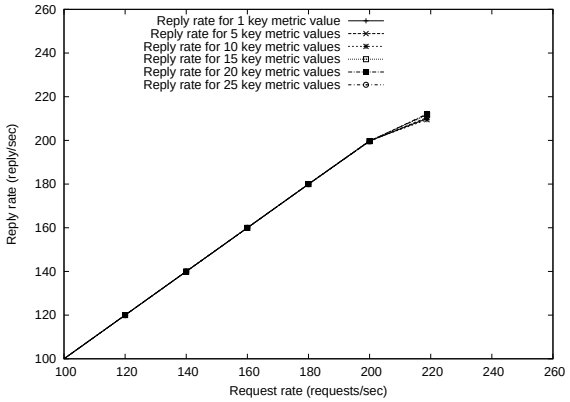

Figure 14: No effect of number of key values

observed reply rate levels off. Also as the complexity of the policy is increased from 1 metric to 250 metrics, the server starts saturating at about 180 requests/sec. Assuming that an average policy consists of 10-20 composite metrics and each metric requires 5-8 computations, our system can support a request rate of about 200 requests/sec even for complex policies.

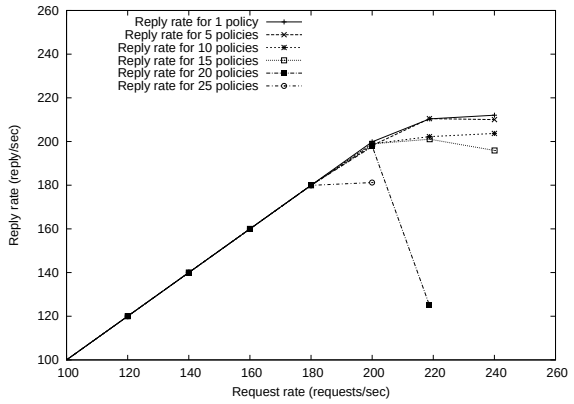

Figure 13: Effect of number of policies

Figure 13 shows a similar plot for increasing number of policies from 1 to 25. Here, the server saturates at about 200 requests/sec for small number of polices, whereas it saturates at 180 requests/sec for 25 policies. Hence, the system is scalable enough to support 180 requests/sec for large number of consumers.

Figure 14 shows a similar plot for increasing number of key metric values in the input data set. Here we observe that the server saturates after 200 requests/sec. Moreover, the plots are approximately the same for any number of key metric values. This shows that the system performance is not effected by the number of key metric values in the input data set. This is as expected because one input record belongs to only one key metric value and therefore, only the computation for that key metric value is performed. Since, the computation procedure is the same for all key metric values, there is no change in the performance.

To measure the "capacity" of our experimental setup, we implemented a simple grid service that outputs "Hello World" on invocation. Using experiments similar to the ones described above, we observed that this service saturated at a request rate of about 525 requests/sec.

5 Discussion

In this section we discuss scalability issues and some applications of Metric Service.

5.1 Distributed Architecture for Scalability

Metric Service may be used in high load applications varying from fault handling in complex IT environments to telecom billing. In such environments scale-up can be achieved by deploying multiple cooperating instances. A distributed architecture is feasible for Metric Service since both its input and output streams consist of metrics alone. Therefore, in many cases it is possible to split a policy into fragments or to create multiple instances of the same policy, each of which can be executed by a different Metric Service instance. Four alternatives are possible:

1. **Vertical Partitioning:** If a policy accepts different kinds of metric records as input then its computation can be partitioned so that input record types that generate heavy load are

processed by different Metric Service instances. The policy tree can be split at those nodes which segregate the LeafMetricDirectives of these input records. By distributing the incoming metric records to different instances, this approach helps in scaling up the overall request rate.

2. **Horizontal Partitioning:** By splitting the policy in a layered manner we can scale up computational capacity in situations where a small number of metric directives constitute the bulk of computation load.

3. **Data Source Partitioning:** It applies whenever the fragments of a policy that involve input metrics from different sources are independent of each other. An example includes fault handling application in a data center where the goal is to receive and interpret fault related events from the servers. In such cases, records from different data sources (e.g. sets of servers grouped on the basis of clients etc.) can be sent to different Metric Service instances. It scales the overall request rate.

4. **Random Load Distribution:** This approach involves randomly distributing the input records to multiple servers each running an instance of the same policy. It is applicable in situations where the metrics being computed in those multiple instances can be aggregated later without breaking the semantics of the policy.

5.2 Applications of Metric Service

Metric Service finds applications in a variety of complex management scenarios. A few of them are presented here.

Telecom Mediation: Telecom infrastructures require mediation between business support systems and network elements. This includes collecting, managing and delivering subscriber data, subscriber service data, usage data and billing information. In traditional circuit-switched networks this information is reported in a Call Data Record (CDR) which has a simple format [14]. It includes fields such as calling and called numbers, local and remote node names, date and timestamp, etc. Since the CDR format was standardized, the same billing software could work across different network products, providers and solutions. However, for IP based networks more information is required to rate the service and content used in IP services. Also, not all such information is relevant to every IP service and newer IP services may require new fields to be introduced. Therefore, a new extensible record format has been proposed by IPDR.org [1]. However, now the content of the IPDRs is no longer fixed and a single billing system cannot be expected to handle all existing and new IPDRs that shall emerge with new IP services. Metric Service can help such systems achieve customizability to adapt to newer IP services.

Distributed Unix Accounting: In the UNIX accounting example we saw that MS-Policy could be used to specify the mediator for accounting process of a UNIX server. This can be extended further to a cluster of servers running different UNIX variants such as Linux, Solaris, AIX[5], etc. Assuming a cluster-wide notion of *user-id*, the same mediator could be used to perform accounting for the whole cluster. The wrappers on each of the machines would map the *pacct* variants on those systems into a common *pacct metric record*. The wrappers could further be enhanced to report server details as well. Cluster-wide accounting enabled in this way could be used for capacity planning, quota enforcement and resource allocation kind of decision making applications. The accounting mediator specification could also be used to augment more general purpose accounting systems that require metrics composition functionality [3].

Web Service Management: Web Services have recently emerged as the technology of choice for building distributed applications. However, their success depends upon the availability of middleware infrastructure for management tasks. Various such middleware components that require derivation of information from the available raw instrumentation data can benefit from the Metric Service engine. Such management components include provisioning, SLAs [8], metering and accounting [2] etc.

6 Related Work

XML and similar languages have been used for mediation, information integration and information exchange. [16] describes a semistructured Object Exchange Model (OEM) which is similar to XML. The OEM model forces no regularity on data and each object has an associated descriptive label that represents its schema. The authors describe OEM's suitability in mediating over heterogeneous databases and

[5]Linux is a trademark of Linus Torvald, Solaris is a trademark of Sun Microsystems, Inc. and AIX is a trademark of IBM Corp. in the U.S., other countries, or both.

Internet sources. Unlike OEM, we do not allow metric records with arbitrary nesting that varies from instance to instance. Instead such nesting is normalized, by wrappers, into multiple metric records each conforming to the same definition. The simplicity of this approach coupled with the ability to define new metric records makes our system applicable to a wide variety of applications.

The MedMaker [15] system describes a mediator specification language that uses declarative rules to describe the functionality of a mediator. Using those rules mediators can be generated automatically. However, their language focusses on mediators that change the view of data, expressed in OEM, from one format to another. This enables the system to present a common view of data to the user.

XMF [10] is an XML based mediation framework that is aimed at providing uniform view of data from heterogeneous Internet data sources such as online stores. They provide a mediator architecture, a wrapper architecture and XML based mediation rules to accomplish this. [18] proposes a new operation called Merge that operates on streams of XML documents, with no schema specified, and combines them into a new XML document. XML Data Mediator (XDM) [20] is a lightweight mediator for bi-directional data conversion between XML and structured data formats such as relational or LDAP data. Most of the systems described above have been aimed at a mediator that selects, restructures and merges information from multiple autonomous sources or sites. This is done for exporting an integrated view of the heterogeneous data for enabling applications such as virtual shopping malls, virtual agencies etc. [4]. In contrast, Metric Service uses the mediator architecture and the flexibility of XML to specify mediator functionality that enables decision making applications to obtain/derive desired information from heterogenous data sources.

Web Service Level Agreement (WSLA) project[6] has developed a framework that allows specification and monitoring of SLAs for web services[8]. MS-Policy has constructs inspired from and, in some cases, directly adopted from the WSLA language. However, certain key differences distinguish the two systems. While WSLA is focussed towards specification of SLAs, Metric Service provides a more generic mediation layer applicable to various management applications. It deals with metrics in conjunction with their associated context information that enables it to handle multiple producers and consumers. Absence of explicit context, in WSLA, allows only one value of a metric to be represented at a point of time and aggregation of metric values cannot be grouped on the basis of some key. WSLA also does not have support for event correlation.

[6] http://www.research.ibm.com/wsla

7 Conclusion

We described a configurable middleware component – Metric Service – that provides a mechanism for declaratively specifying and executing the logic for creation of information required by management applications. We introduced – MS-Policy – an extensible XML based language that allows specification of the information derivation process in terms of metrics aggregation and composition. We described the system design and demonstrated it through a prototype implementation. Metric Service allows information manipulation in addition to filtering allowed in publish-subscribe systems and also handles traditional information integration issues. We demonstrated that Metric Service enables concurrent execution of multiple policies making it shareable among multiple management applications simultaneously. We evaluated our implementation and discussed how it can be scaled up further for high load environments.

The ability to specify a management task declaratively can be of great importance in management frameworks such as Distributed Management Task Force's CIM[7]. This is mainly because such frameworks enable a common management infrastructure for the whole IT environment but lack the ability to provide management functionality in the middleware itself. Metric Service can help fill some of that gap and we are investigating it further.

8 Acknowledgments

We thank Dr. Neeran Karnik, IBM India Research for his meticulous and insightful reviews, Dr. Bill Horn, IBM T. J. Watson Research for introducing us to this problem and Dr. Sugata Ghosal, IBM India Research for his support.

References

[1] Network Data Management-Usage For IP-Based Services v3.1.1. http://www.ipdr.org, Oct 2002.

[2] V. Agarwal, N. Karnik, and A. Kumar. Metering and Accounting for Composite e-Services. In

[7] Common Information Model. http://www.dmtf.org

Proceedings of the IEEE International Conference on Electronic Commerce (CEC), CA, June 2003.

[3] V. Agarwal, N. Karnik, and A. Kumar. An Information Model for Metering and Accounting. In *Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS), Korea*, April 2004.

[4] C. K. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. XML-Based Information Mediation with MIX. In *Proceedings of the ACM SIGMOD International Conference on Management of Data,USA*, June 1999.

[5] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, 35(6):37–46, June 2002.

[6] S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, and R. Neyama. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*. Sams, 2001.

[7] V. Hazlewood, R. Bean, and K. Yoshimotoe. SNUPI: A grid accounting and performance system employing portal services and RDBMS back-end. In *Linux Revolution Conference, NCSA,Illinois*, June 2001.

[8] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1), March 2003.

[9] A. Kumar, V. Agarwal, and W. P. Horn. Metric Service Policy Language Specification Version 1.0 (MS-Policy). IBM Corporation, December 2003.

[10] K. Lee, J. Min, K. Park, and K. Lee. A Design and Implementation of XML-based Mediation Framework (XMF) for Integration of Internet Information Resources. In *Proceedings of the 35th Hawaii International Conference on System Sciences*, 2002.

[11] J. P. Martin-Flatin. Distributed Event Correlation and Self-managed Systems. In *Proceedings of SELF-STAR : International Workshop on Self-* Properties in Complex Information Systems, Italy*, June 2004.

[12] Common Information Model (CIM) Metrics Model, v2.7. Distributed Management Task Force, http://www.dmtf.org/standards/documents/CIM/DSP0141.pdf, June 2003.

[13] D. Mosberger and T. Jin. httperf–A Tool for Measuring Web Server Performance. In *Proceedings of Workshop on Internet Performance, USA*, June 1998.

[14] G. Noisette. IP Billing. HP Intel Solution Centre. http://www.hpintelco.net/pdf/solutions/telecom/operations/IPBilling_techb.pdf.

[15] Y. Papakonstantinou, H. Garcia-Molina, and J. D. Ullman. MedMaker: A Mediation System Based on Declarative Specifications. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE)*, pages 132–141, 1996.

[16] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE), Taipei, Taiwan*, March 1995.

[17] D. Powell. Guest Editor. Group Communication. *Communications of the ACM*, 39(4):50–97, 1996.

[18] K. Tufte and D. Maier. Aggregation and Accumulation of XML Data. *IEEE Data Engineering Bulletin*, 24(2):34–39, 2001.

[19] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer Magazine*, 25(3):38–49, March 1992.

[20] N. Zhou, G. Mihaila, and D. Meliksetian. XML Data Mediator - Integrated solution for XML Roundtrip from XML to Relational. In *Proceedings of the Thirteenth International World Wide Web Conference, New York City, NY*, May 2004.