# The utility metering service of the Universal Management Infrastructure

by V. Albaugh
H. Madduri

One of the main characteristics of on demand computing in general and of utility computing services in particular is the "pay-as-you-go" model. To implement this model, one needs a flexible way to meter the services and resources being used. The UMI (Universal Management Infrastructure) architecture, designed to provide common functions that are needed by most, if not all, of the utilities in a utility computing system, therefore includes a metering function. The architecture of the metering system is hierarchical and highly flexible. This paper reviews the metering service architecture and describes how UMI's metering service function is used in the context of utility computing services, for collecting and storing metered data, computing service metrics (which are useful to the data-consuming applications), and feeding the metrics to various consumer modules (e.g., for accounting and billing).

Utility computing brings the benefits of reduced IT (information technology) complexity, variable pricing, and reduced operating costs to enterprises.[1,2] Utility services can be infrastructure- or application-level services that are sold on a "pay-as-you-go" basis. To emphasize the on demand characteristic of these services, we refer to them in this paper as "on demand services" (ODSs). This is intended to be a generic term for the technical community and is not to be understood as the name of a product offering or marketing initiative. We describe the architecture of ODSs and the process of their development and present our vision for metering services for ODSs that use the Universal Management Infrastructure (UMI).

**ODS architecture.** On demand services (ODSs) form a layered hierarchy. *Infrastructure* ODSs, used by all the higher layers, are at the base. Above these are the *application-level* ODSs.[3] Both of these layers can be further divided into *common* and *specialized* ODSs. For example, a metering or monitoring ODS can be thought of as a common infrastructure-level ODS, whereas a firewall service can be thought of as a higher-level specialized infrastructure service. Likewise, a billing service is an example of a common service at the application level, whereas a supply chain management application is a specialized application-level ODS.

All ODSs use UMI for certain basic services. UMI is both a platform for flexible service delivery and a management discipline to automate the data center.[1] UMI watches the general health of ODSs running on it and provides a stable running environment for them. Our vision for the future of UMI is that after an ODS is loaded, configured, and running in the UMI environment (which consists of a hierarchy of ODSs plus the UMI base), the ODS should not have to ask UMI for any resources or for any specific help. Instead, UMI will monitor the ODS, notice what it needs, and provide the additional resources or corrective actions, as specified by some predefined service management policy. To accommodate the phased implementation of this vision and to allow for dynamic and flexible control by the ODSs as well,

Table 1 Definition of terms

| Term | Definition |
|---|---|
| Variable | A thing of interest whose value can change over time. |
| Sensor | An instrument that captures a variable's value at a given time. |
| Measurement | An observed value of a variable. |
| Metric | A value that is arrived at by computation, as opposed to a measurement (q.v.) that is arrived at by observation. It typically uses a formula defined by the ODS owner or UMI administration (e.g., service unit computation). |
| Agent | A collection of sensors on a given server (one may also look at it as the single conduit over which all sensors on a server send information). |
| Collector | A consolidator of measurements received from all agents in a data center or data center region. |
| Metering engine | A system which computes metrics (q.v.), refines and stores measurements and metrics, and provides metrics to consumers. |
| Metrics consumer | A subsystem that establishes a connection and protocol with the metering engine for the timely transfer of metrics data (e.g., usage in service units [q.v.] for billing purposes). |
| Metering record | Refers to the format that is used to communicate and store a single measurement. |
| Resource identifier | Refers to the name given to a specific variable. It is used as a field in a metering record to identify the variable that the measurement is reporting. |
| Service unit | Refers to a measurement or composition of measurements that reflects usage and the business model a service uses to charge their customers. Customers are charged for the "service units" they consume. |
| "Virtual" | As a modifier for record, resource, or service unit (q.v.), "virtual" indicates that it is formed through composition rather than captured as a measurement. |

UMI is expected to provide some well-architected service programming interfaces (SPIs). ODSs can use these to request UMI services, to alter configurations, or to get information in and out of UMI.

In order for higher-level ODSs to take advantage of lower-level ODSs, there must also be interfaces and protocols for ODSs to talk to one another. It is expected that these interfaces and protocols will be based on Web-service standards (XML [eXtensible Markup Language], WSDL [Web Services Definition Language], SOAP [Simple Object Access Protocol], and UDDI [Universal Description, Discovery, and Integration]). It is expected that a higher-level ODS will serve its customers by occasionally calling lower-level services for the lower-level resources and services that it needs to satisfy its customers. For example, a Web e-mail ODS may need 5 GB of additional disk space to accommodate a request to create 1000 new e-mail accounts, and the ODS would request this from an infrastructure ODS that allocates disk storage. The Web e-mail ODS may want to meter e-mail accounts for their usage, whereas the disk storage ODS may want to meter storage usage (and potentially charge its respective customers based on usage). Both services could take advantage of UMI metering independently, and possibly for completely different purposes (e.g., one service might meter for billing purposes, whereas the other might meter for optimizing performance).

Although much of the discussion of metering applies equally to both infrastructure ODSs and application-level ODSs, in the following sections we will focus on application-level ODSs only, unless otherwise indicated.

**The ODS development model and process.** The development of an ODS on top of UMI must adhere to certain guidelines and standards, which will be published by the UMI development team. These guidelines and standards lead to a development process that systematically specifies what changes an application has to undergo in order to become a well-behaving ODS in the UMI environment. (IBM's Application Enablement Program [AEP][4] team helps software vendors adhere to this process for faster and easier deployment.) We will not go into the details of all of these changes, but we will discuss what the ODS needs to be aware of with respect to metering in the section "Metering architecture." Table 1 presents some concepts and definitions that will be used in the presentation of the metering architecture and functions.

**The importance of metering.** Metering is an essential function in the world of on demand computing.

Without a flexible and generalized metering function, the on demand vision cannot be realized. An on demand business, by definition, must be focused, resilient, and robust in response to changing conditions. Metering is essential in implementing systems that support variable cost, resiliency, and responsiveness. Autonomic behavior, which is a fundamental technical characteristic of on demand systems, depends upon implementing a closed-loop control system, at the heart of which is metering.

Metering data is critical because loss of this data can mean a loss of revenue. Some specific examples of the use of metering in on demand systems and solutions include usage-based billing, "charge back" to user departments of a consolidated service, capacity planning, and studying end-user usage patterns for improving customer service or inventing new services.

**UMI.** UMI is an infrastructure that supports utility computing systems. Some of the underlying concepts of UMI are similar to those underlying a phone company infrastructure, which supports services like local and long distance calls, Internet service, and DSL (digital subscriber line) service. The infrastructure has enough capacity to cope with variation in demand and provides stability for the services. The vision of UMI is to build an environment that provides stability for the ODSs by coping with fluctuating resource needs.[5] It is also meant to promote autonomic behavior of the ODSs so that when they deviate from expected behavior or performance, they can be corrected and brought back to normalcy. This correction is effected by constant monitoring of ODSs and application of prespecified policies when conditions deviate. This kind of autonomic behavior is referred to as *policy-based management* in UMI.[6]

There are roughly a dozen components which are expected in the initial release of UMI. The main ones are metering, monitoring, auto-provisioning, SLA (service level agreement) management, portal, billing, ordering, reporting, and helpdesk/change management. The capabilities of UMI are described on IBM's Internet Web site and elsewhere in this issue.[3,6]

Metering is a key component of UMI and interacts with the UMI components that manage billing, reporting, and monitoring. The metering function of UMI is designed to handle usage metering of both system-level resources and application-level resources as the application services (i.e., ODSs) are exercised by their end users.

In the remainder of this paper, we discuss the role and implementation of metering services generally and their importance in the provision of ODSs. We review the metering service architecture and describe the metering service function of UMI (and its benefits) in the context of utility computing services.

It should be noted that the descriptions of UMI, metering architecture, and its implementation that are presented here should be viewed as intended directions and not as commitments to be delivered in IBM products. The actual functions and features delivered in UMI releases may vary considerably from what has been described or referenced in this paper.

## Metering

Requirements for a metering service pertain to the metering agent and the metering engine. Flexibility is a pervasive requirement. Of course, there are also general requirements like scalability, reliability, and availability that apply to the metering service as a whole. Table 2 summarizes these requirements.

In addition, it is useful to distinguish between metering and monitoring[7] functions. Table 3 presents the main distinctions between these two functions, emphasizing that metering, in contrast to monitoring, typically involves observing usage of a resource without interpreting the usage, tying those observations to a user or account, or retaining them for purposes of auditing.

Finally, one may observe that there are similarities among metering, monitoring, and SLA (service level agreement) management components. They all depend on raw measurements (or observations)—in fact, the "sensors" that provide these raw measurements could be the same for all three. The computations, transformations, and threshold logic applied to these measurements differ in each component, but they all have the same basic design pattern. The pattern has three parts: (1) collect the raw measurements, (2) process them in flexible ways (a processing engine), and (3) consume the processed results in order to effect changes in the larger system. This design pattern is shown in Figure 1. In the following sections we will expand on this basic design pattern as it applies to metering.

Table 2 Metering requirements

| Types of Requirement | Description |
|---|---|
| Metering agent requirements | Ability to gather data from various devices (each device type has its own sensor type)<br>Ability to poll for data as well as accept pushed data<br>Ability to collect data on a timer<br>Metering physical resources vs logical resources (virtual resources)<br>Metering application-level events<br>Flexibility at the input end (measurements) and at the output end (metrics)<br>Small footprint and good performance |
| Metering engine requirements | Flexible metrics calculations (measurement aggregation, composition, and other derivatives)<br>Flexible ways to feed metering metrics to different consumer modules/services<br>Data retention for reliability and audit purposes<br>Agent-to-engine and engine-to-consumer data transfer protocols to include audit scopes<br>Flexible reporting of raw measurement data as well derived metric data |
| General metering requirements | Scalability, reliability, and availability<br>Correctness, consistency and auditability<br>No loss of data in transit<br>Ability to start and stop components independently<br>Ability to have sensors and the engine in different network segments and even geographically separated<br>Secure data communications between agents, collectors, metering engine, and consumer modules<br>Manageability |

Table 3 Characteristics of monitoring versus metering

| Monitoring | Metering |
|---|---|
| Makes observations. | Metering also makes observations, but the data collected and the frequency of collection might be different. |
| Main function is checking for something (i.e., comparing against some predefined values or conditions). | Main function is counting. |
| Function includes rules to interpret and judge what is being observed. | Function usually does not interpret or judge what is being observed—this is performed by other modules that consume the metering function's output. |
| Typically, data is summarized and original observations are discarded quickly. | May have to retain original observations for auditing purposes (for example, data that drives billing). This has implications for both storage and data transmission. |
| Monitoring output for a shared resource typically does not tie measurements to a user or account. | Metering a shared resource typically ties measurements to a user or account. |
| May have more stringent real-time requirements than metering. | May have real-time requirements also, but typically not as stringent as those of monitoring. |

## Metering architecture

In this section, we describe the entities of which the metering architecture is comprised and the interface protocols that the architecture supports.

**Entities.** The entities comprising the general metering architecture have already been introduced and described briefly in the definitions of terms given in Table 1. They are discussed here in further detail and in the context of the overall metering architecture. The architecture provides for a hierarchical collection of data from the endpoints to collectors and from there to the metering engine, as shown in Figure 2.

***Sensor.*** Sensors capture measurement values for variables of interest. These measurements are an indication of resource usage or event occurrence. Sensors are generally customized code implemented to capture a particular variable's value. There are many

ways in which they may be implemented. They may be embedded in the application code for a service or implemented externally to the service code. They may sample allocation of resources periodically, or they may track every allocation and deallocation of resources; they may execute system commands to capture system resource usage; or they may scan log files or databases to extract measurement values. Sensors record their measurement values by providing them to the local agent, which takes responsibility for the data upon receipt. There is one or more sensor for each endpoint.

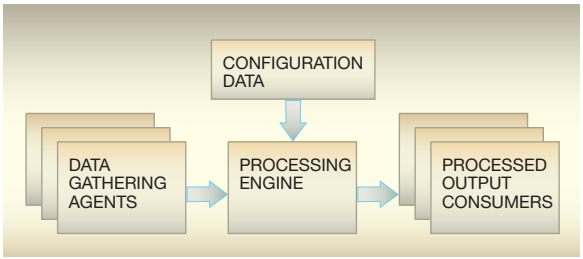
***Agent.*** There is an agent at each endpoint. The agent can be viewed as the single conduit for the local sensors to forward their measurements. It presents a standard local interface to each sensor. The agent accumulates and buffers the measurements and forwards them to the collection process. The agent takes responsibility for the data on receipt from a sensor and must retain the data that it is buffering for purposes of recovery from failures such as a system power outage.

***Endpoint.*** An endpoint is simply any system that is a source of metering data. It has an agent and one or more sensors. Any ODS that is providing metering data will have one or more endpoints. It is also possible that each system allocated to an ODS (e.g., an e-learning service ODS) from some other provisioning ODS (e.g., a virtual partition provisioning service ODS) has sensors and an agent which are in place already to support metering of the provisioned resource (in this case, the use of the virtual partition).

***Collector.*** A collector exists in the architecture primarily for scaling and redundancy. It is a conduit for data from agents to the metering-engine data storage. As more endpoints are added in the network, additional collectors can be added as needed. They provide for parallel processing of data before it is recorded in the metering-engine data storage. Multiple collectors provide an alternate path for data if one collector fails.

***Metering engine.*** The metering engine conceptually is a single system but may be implemented across multiple systems as dictated by the size of the supported network. As shown in Figure 3, it houses the database for storage of metering records. The database can be partitioned by the ODS to avoid contention and split across multiple servers for scaling as necessary. It contains the configuration data for each ODS, which includes the rules for scaling, filtering, and composing metering records into virtual service units. The metering engine performs the data processing indicated by the configuration of the ODS. The metering engine also provides an interface for consumers of the processed metering data to request and receive specified subsets of data.

Figure 1 Basic design pattern for metering, monitoring, and SLA management components




***Consumer of metrics.*** There are multiple consumers of metering data. A primary consumer of metering data is a billing system, which applies rate plans to the data and prepares customer invoices. Other consumers of the data include data mining services and report services that may further process and present the data for analysis and response. The ODS may request data for various reasons, including gathering real-time usage statistics for presentation to end users on demand.

**Interface protocols.** The architecture can be supported with various interface definitions for the agent, collector, and metering engine. It is important that the interface be defined such that the receipt of data is explicitly acknowledged and the entity accepting data becomes responsible for retaining the data until it is transferred.

## Mapping to implementation technologies

The metering architecture described in the previous section can be implemented in many ways, which involve many design choices. This section offers guidelines for implementing a metering service for an ODS and its interfaces and gives examples of expressions used in creating virtual records by means of composition rules.

**Architecture vs implementation in UMI.** The UMI implementation environment imposes constraints that affect the implementation and realization of the me-

Figure 2 Metering architecture

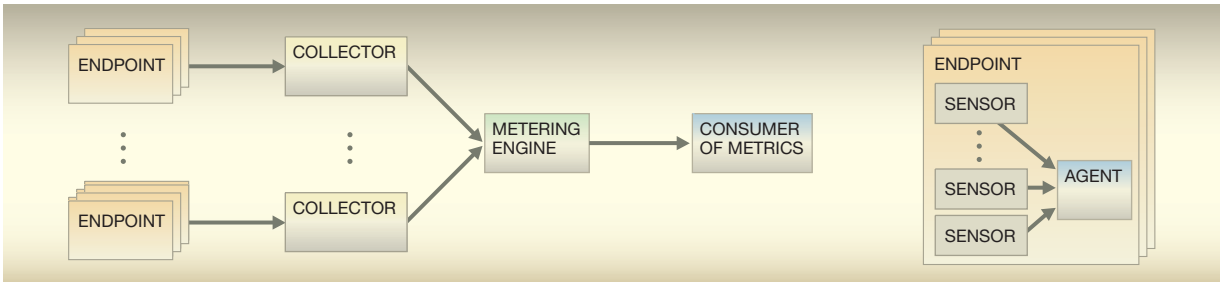
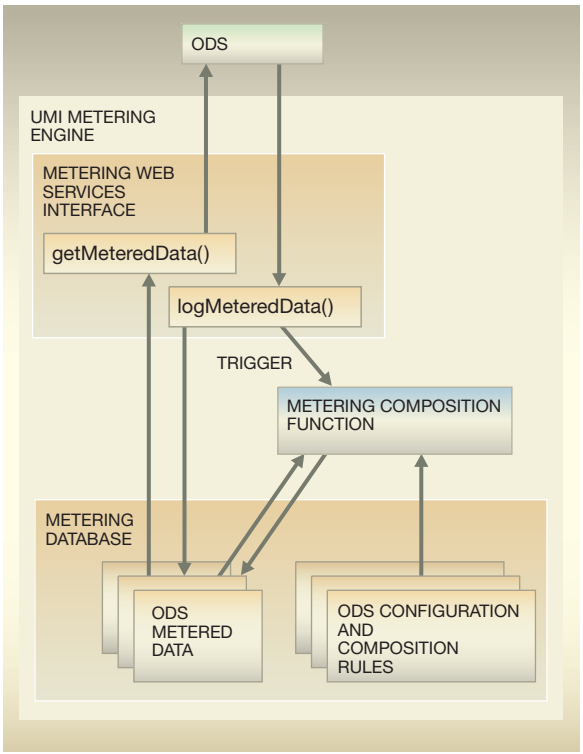



Figure 3 UMI metering engine architecture




tering architecture. The primary constraints affecting the implementation are:

(1) The endpoint is the domain of the ODS. The sensors and agents are in the endpoint owned by the ODS. UMI does not require ODS systems to host a specific implementation of a metering agent. However, UMI intends to provide a sample implementation of an agent that may be used by an ODS either directly or with needed modifications. With no assurance of common agents at endpoints, the implementation does not support the concept of centralized configuration or management and control of agents. Instead, the interface to the collector is defined as a callable interface that endpoints invoke to get or log metering data.

(2) UMI presents its services to ODSs through a Web Services interface using SOAP remote procedure calls. This requires UMI metering to present the interface to the collector as a Web Services call. This call, logMeteredData(), is described in detail later in this section. The collector function in the architecture is realized in UMI through the implementation of this Web Services call, which inputs data to the metering-engine data storage. The ability to have multiple Web servers to present the interface fulfills the scalability requirement of the collector in the architecture.

The consumer interface is also implemented as a Web Services call, getMeteredData(). It allows consumers to request processed metering records and is also described in detail later in this section.

**Architectural entities implemented in the ODS.** The ODS wishing to record metering data must implement the logical functions of an endpoint. This includes the implementation of sensors and an agent, as shown in Figure 4.

***Sensors.*** An ODS must decide what variables it wishes to measure and assign resource IDs to the variables. The ODS must then decide in what units to express the variables for measurement and how it will instrument its service to obtain values for those variables. The implementation of that instrumentation is the responsibility of the sensors in the ODS.

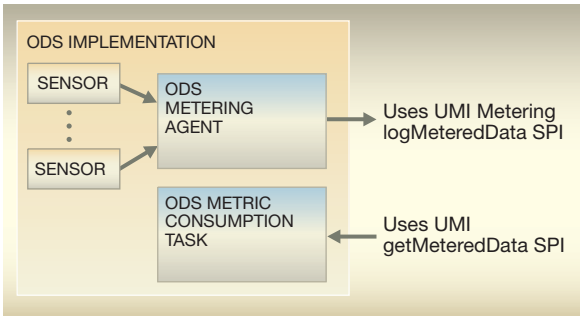
As an example, let us assume the ODS is an income tax preparation Web application, and the following variables are to be measured:

- The length of time a user is logged into the service in seconds ("sessionTime")
- The number of forms the user accesses each session ("formsAccessed")
- The number of help menus the user accesses each session ("helpAccessed")
- The number of completed forms contained in the final Internal Revenue Service submission ("formsSubmitted")

There are many ways the ODS might choose to instrument (i.e., build sensors for) any of these variables. Let us consider sessionTime. The ODS may choose to sample the time when a user logs in, sample the time again when the user logs out, and submit one metering record for sessionTime at log out. An alternate implementation might be to submit a metering record every 30 seconds while the user is logged in. With the latter approach, the ODS can maintain more current data in the metering engine and exploit the composition functions (such as simple aggregation) in the metering engine to generate the user's total sessionTime on demand. It is expected that many ODSs will want to record metered data in real time, exploit the composition functions in the metering engine, and exploit the consumer interface to get processed records back on demand. But ODSs may use other mechanisms. For example, an ODS may be an existing application that has always recorded usage statistics in its own database or log files. When adapting to UMI, it may choose to scan those local repositories once a week or once a month to provide its metered data. The choice is entirely up to the ODS.

A question that might arise here is "How is this architecture applicable to traditional hardware measurements like CPU seconds, bytes of memory, disk storage, etc.?" Traditional measurements can be handled in the following way. One can build sensors for these hardware usage measurements, which would simply be wrappers on the corresponding operating system functions, and connect them to the agent running on the corresponding hardware platform. The sensors are governed by the agent. By the time the measurements leave the agent and get into the collector, they look no different from the software ODS measurements mentioned above.

Figure 4 Logical functions for ODS metering implementation




***Agent.*** The requirements for an agent implementation in the ODS are driven by performance. The overhead of invoking a remote Web Services call dictates that the collector interface be defined to accept an array of metering records. For a service that wishes to submit records in real time, this is most easily achieved through the implementation of an agent that accepts records in real time from the various sensors, buffers them, and forwards them to the logMeteredData() interface in batch. (It may appear that these calls could be asynchronous, but that makes the programming somewhat harder due to reliable delivery considerations.) UMI intends to provide sample code for such an agent that ODSs may use, modify, or emulate.

The implementation of an agent must include the interface it will expose to the sensors for receiving metering records. It must also include the policy that it will enforce regarding when to forward buffered data to the logMeteredData() interface. This could be configurable based on a combination of time and the amount of data, such that it would, for example, forward buffered data when the number of records reaches a count of 500 or when five minutes elapses, whichever comes first. The agent could also expose a "flush" interface that is used by other parts of the application to flush any buffered data on demand to the collector, as when a consuming part of their application requests processed data back from the metering engine through the getMeteredData() interface.

***Consumer.*** An ODS may invoke the consumer interface, getMeteredData(), for various reasons. The example already given is that the ODS may want to present up-to-date usage statistics to a user on demand. Alternatively, the ODS may also want to ac-

Figure 5 Java signature of logMeteredData() method

```
public LogMeteredDataResponse logMeteredData(
        String odsName,                 // Identifies which ODS is invoking the call
        String auditScopeId,            // Used for data reconciliation between client and
                                        // server; = ODS source identifier + YYYYMMDD
        int packetId,                   // Uniquely identifies specific invocation of services call
        MeterData[] meterData,          // An array of MeterData objects, each with instance
                                        // variables resourceId, resourceUserId,
                                        // measurementValue, and unitOfWork and array of Attribute objects
        int packetRecordCount,          // Indicates how many records client has declared
        float packetHashValue,          // Sum of all measurementValues in the call
        int auditScopeRecordCount,      // Running sum of all records sent within audit scope
        float auditScopeHashValue       // Running sum of packet hash values in audit scope
)
```

cess, for a given time period, billable records for all users, to feed an invoicing service. It may request other non-billable records that it has instrumented solely for the purpose of analyses of usage patterns.

**Architectural entities implemented in the UMI.** The collector and consumer interfaces for metering are implemented as Web Services calls in UMI. The metering engine database logically includes the configuration data and composition rules for each ODS. The ODS metering data includes both the records that are explicitly recorded through the collector interface and the virtual records that are calculated based on composition rules configured for the ODS.

**The collector interface—logMeteredData().** The Web Services call is implemented through the logMeteredData() method of a Java class. The caller inputs an array of MeterData objects as one of the parameters to this call. Each MeterData object represents a record of metered usage. The Java signature of the logMeteredData() method is shown in Figure 5.

Of particular importance is the unitOfWork parameter. This parameter is used to define a processing boundary for any specified composition rules, including aggregation. This allows the ODS to associate multiple metering records with a specified unit of work. The unit of work value is generated by the ODS and must be unique, that is, different from any previous specified unit of work value the ODS has used. A unit of work is opened with the first record received that references the unit of work value and closed when a record is received referencing the unit of work and having the closeUnitOfWork field set to true.

The metering engine provides the capability to combine multiple records into a new calculated (virtual) record based on the composition rules configured in the metering engine for the ODS. For this processing to occur, it is necessary to identify the record set to be processed. The unit of work defines a set of records for which composition rules can be applied. When the unit of work is closed, composition processing is triggered. A unitOfWork value of zero is reserved for records that are not intended for composition. This value may be used at any time for any record and simply excludes such records from selection in any composition formulas. Such records are immediately available for downstream processing.

As one example of using the unitOfWork parameter, an ODS may want to bill based on hourly usage. In that case, the ODS may put all records for an hour in a single unit of work, closing the unit of work and starting a new unit of work each hour. Each time a unit of work is closed, the composition rules (in this case, simple aggregation) would be applied to the set of records in the unit of work. As another example, an ODS may want to bill based on usage for a login session. In this case the ODS may start a unit of work when the user signs in to the service, record all usage statistics for the user while signed in to that unit of work, and close the unit of work when the user terminates the session. Real and virtual resource records for that unit of work would be processed when the unit of work is closed.

**The consumer interface—getMeteredData().** This interface is intended for getting the metering data out of the metering engine and into a consuming module or service. This consumer could be a billing service or a reporting service or even the ODS itself. The function of this Web Services call is implemented through the getMeteredData() method of a Java class. The call is presented in a manner similar to a database query, where the metering records returned are those which satisfy the selection criteria. The Java signature of the call is:

```
public GetMeteredDataResponse
getMeteredData(
    String transId,
    String odsInstanceId,
    SelectionCriteria SC
)
```

The SelectionCriteria is a container class that specifies the constraints of the query. These constraints include unitOfWork, startTime, stopTime, resourceId, resourceUserId, and startingRecordReference. Any of these constraints may be specified or left at their default values. Only the constraints specified are applied to the query. There is a configurable limit on the number of records returned in any response. The records returned are sorted in order of the record reference identifier. If the number of records satisfying the query exceeds the maximum that can be returned, only a partial set is returned, and the caller may retrieve the remaining records by reissuing the query with an adjusted value for startingRecordReference. The response class contains, among other things, the array of meter records being returned and a flag indicating whether all the selected records are contained in the response.

**Composition and virtual records.** Composition is the process of creating new metering records, referred to as virtual records, by selecting and combining real metering records. The new records are added to the database and can be selected like any other record through the consumer interface. In order to perform composition, the following need to be defined:

- Set of records to be considered in the selection process
- Criteria for selecting records to be included in composition
- Calculation to be performed in computing the new measurement value
- Attributes to apply to the newly calculated record

The following sections describe how these parameters are defined in UMI metering.

***Defining the set of records for processing.*** The set of records to be considered for processing must be communicated from the ODS. If the metering engine were to arbitrarily initiate composition processing, this might result in an incomplete record set that has only a subset of the records intended for inclusion in the composition. The mechanism used to communicate these processing sets is the unit of work. Every metering record is tagged as belonging to a unit of work. Composition processing is triggered when a unit of work is closed by a "sentinel" record that belongs to the unit of work but also indicates that the unit of work should be closed.

An ODS that does not define composition rules can simply send the zero value for unitOfWork with each record and not be concerned with closing the unit with sentinel records. Similarly, an ODS that has only defined simple aggregation for its composition rules can reuse the zero value of unitOfWork and simply use a sentinel record to define the aggregation boundaries. For example, if the ODS wants to create hourly aggregation records, it can send a sentinel record every hour. The granularity on defining the unit of work is left to the ODS. An ODS that presents itself as a Web application may want to create a unit of work for each user session and perform processing when the session terminates. The mechanism is very flexible and can support a very wide range of usage.

***Defining the records to be selected in a composition rule.*** A record or records must be selected in a composition rule by the resource identifier. The selection may be further qualified by specifying values for additional fields and key/value attribute pairs contained in the record. For the purposes of discussion, we can think of the selection rule as being expressed as

```
Selection X = <resourceId=value,
resourceUserId=value, key/value,
key/value, ..., key/value>
```

where *X* is the record or set of records selected by the specified criteria. If the selection criteria that is specified selects a set of records, a simple aggregation is performed, resulting in a single record that maintains common field values. The only required field in this record selection format is the resource identifier. UMI metering reserves the key value of

"UMI_unique" to imply multiple execution of the rule for each unique value of a field. For example, the following composition rule fragment selects and adds the formsAccessed and helpAccessed resource identifiers that are marked with the same value for the session key, and would form a new resultant virtual record for each session in the unit of work set:

```
<resourceId=formsAccessed,
"Session"/"UMI_unique"> +
<resourceId=helpAccessed,
"Session"/"UMI_unique">
```

***Defining the computation in a composition rule.*** The composition rule defines how the value for the new resource identifier is computed. Aggregation and linear arithmetic on the metric values can be supported. Aggregation is expressed by enclosing brackets. The rules format and description is most readily described by example. The following expression would define a virtual metering record that has the virtual resource identifier of sessionFormsAccessed:

```
<Virtual
resourceId=sessionFormsAccessed> =
<resourceId=formsAccessed,
"Session"/"UMI_unique">
```

It is calculated by the summation of all the records for a user with a resourceId of formsAccessed that are marked with the same value for the session key.

Linear expressions for calculation of values can be formed as follows, for example:

```
<newResourceId> = 2<aaa> +
0.5(<bbb>) + 10<ccc>
```

This expression accesses the record defined by <aaa> and scales it by 2, adds the intermediate result with aggregated records defined by <bbb> and scaled by 0.5, and adds that intermediate result to the record defined by <ccc> and scaled by 10.

***Defining the attributes to apply to a virtual record.*** The virtual records created by aggregation or linear composition can be populated with static values defined in the rule or with "key/value" pairs used to qualify records in the selection portion of the rule. If we use the syntax <new,"keyname", "keyvalue"> to indicate a new attribute with static values, and the syntax <resourceId, keyname> to reference attribute values from a selected record, then we can extend our previous example for an ODS that sends a sentinel record every 30 minutes as follows:

```
<sessionFormsAccessed,
<formsAccessed,"Session">, <new,
"aggregationPeriod", "30 minutes">> =
<formsAccessed, "Session"/"UMI_unique">
```

It creates a new sessionFormsAccessed record that has the session attribute and value from the reference and adds a new attribute which sets the aggregation period.

## Benefits of metering

The metering system architecture presented here offers flexible ways to meter almost anything. The ODS owner simply needs to build a sensor and connect it to the agent at the endpoint. Raw measurements can be flexibly composed into service units or other metrics of business value. Computed metrics can be provided to a variety of consumers through simple interfaces and protocols.

The protocol among metering agents, collector, and the engine that keeps track of audit scopes, record counts, and so forth, allows for full compliance with business systems' audit standards. The key/value parameters passed to the metering system allow the usage to be labeled with identifiers specifying end users and contexts, allowing for fine-grained analysis of usage at any level. This architecture enables extensive analysis of usage and events to improve performance, response times, and customer satisfaction, and true "pay-as-you-go" models of ODSs can be achieved.

Metering is a key function enabling the development of ODSs, and thus positions IBM better with independent software vendors. Metering and billing are the quintessence of the "pay-as-you-go" model of services. Metering is also critical for IBM to assess how well its own hosting infrastructure is doing and to make the necessary capacity adjustments.

## Conclusion and future work

We have presented the architecture and design of a metering service using UMI, one which is very flexible in terms of data input, computing metrics, and providing metric output to consuming modules and services. We have also presented a mapping to the current implementation planned for the UMI 2.1 release.

The future of metering systems can proceed in several directions:

- Implementing all the elements of the metering architecture with well-defined protocols to a variety of producers and consumers of metering data.
- Standardizing and enabling third-party vendors to build sensors and agents for various resources and services.
- Adding instrumentation to middleware like WebSphere* MQSeries* or WebSphere Application Server Enterprise (i.e., Java 2 Enterprise Edition** containers) so that some level of usage metering can be done without having to build instrumentation into the target applications.
- Developing easier ways to specify composition of metrics to be performed—perhaps by using a GUI (graphical user interface) tool (e. g., wizards).
- Developing easier ways to configure metering elements for a new ODS—again perhaps by using a GUI tool.

Regardless of which of these directions (or an alternative) is chosen by the implementors of metering services, it is clear that the role of metering in the provision of on demand services is critical to the success of those services and that this role will continue to develop and expand.

## Acknowledgments

There were many reviewers who helped us improve the presentation of material here. We thank Mike Polan, Jim Adreon, Messaoud Benantar, and many anonymous reviewers who provided feedback to improve the quality of this paper. We thank the UMI architecture team, who contributed to the definition and refinement of metering requirements, and the UMI metering team, whose implementation efforts helped us gain a much better understanding of the nuances of metering in the context of billing.



## Cited references

1. IBM Venture Development—Flexible Hosting Services (Utility Services), http://www-1.ibm.com/mediumbusiness/venture_development/emerging/fhs.html (2001).
2. G. Thickins, "Utility Computing: The Next New IT Model," http://www.utilitycomputing.com/news/258.asp (April 2003).
3. K. Chang, A. Dasari, H. Madduri, A. Mendoza, and J. Mims, "Design of an Enablement Process for On Demand Applications," *IBM Systems Journal* **43**, No. 1, 190–203 (2004, this issue).
4. IBM e-Business Hosting—Application Enablement Program for Utility Service Providers, http://www-3.ibm.com/services/e-business/hosting/mgdhosting/usp.html (2002).
5. "IBM Acquires Think Dynamics", http://www-3.ibm.com/software/tivoli/features/provision/pressrelease.html (May 2003).
6. K. Appleby, S. B. Calo, J. R. Giles, and K.-W. Lee, "Policy-based Automated Provisioning," *IBM Systems Journal* **43,** No. 1, 121–135 (2001, this issue).
7. M. Moeller, E. Manoel, S. K. Mohammed, and M. Parlione, *Introducing IBM Tivoli Monitoring for Web Infrastructure*, IBM Redbook SG24-6618-00, http://publib-b.boulder.ibm.com/redbooks.nsf/RedbookAbstracts/sg246618.html?Open (December 2002).

*Accepted for publication September 14, 2003.*

**Virgil Albaugh** *IBM Global Services 11501 Burnet Road Austin, TX 78758 (albaugh@us.ibm.com).* Mr. Albaugh has a B.S.E.E. degree from the University of Akron in Akron, Ohio and a Masters degree in electrical engineering from the Rensselaer Polytechnic Institute in Troy, New York. He worked at General Electric Company as a developer for the fire control system of the Trident Submarine and is a graduate of General Electric's Advanced Course for Engineers. Since joining IBM in 1976, he has worked on hardware development, chip design, printer development, AIX operating system development, AIX performance tools, 3D graphics, multimedia, WebSphere development, electronic billing services, data transformation, UMI metering, and the Application Enablement Platform for UMI. He is currently the technical team lead for the Application Enablement Platform for UMI.

**Hari Madduri** *IBM Software Group (Tivoli) 11501 Burnet Road Austin, TX 78758 (madduri@us.ibm.com).* Dr. Madduri has an M. Sc. degree in physics from Osmania University (India) and an M. Phil. degree in computer applications from the University of Hyderabad (India). He also has Masters and Ph.D. degrees in computer science from the University of Wisconsin, Madison. Before joining IBM, he was a researcher at Honeywell Inc. working on fault tolerance and distributed systems. Since joining IBM in 1990, he has worked on distributed object technologies, data mining, and hosted service development. His more recent projects have been in electronic bill presentment and payment services, data transformation services, UMI, UMI metering, application framework for UMI, and the Application Enablement Program for IBM server farms and UMI. Almost all of these services are examples of the "software as a service" concept. Dr. Madduri is currently an architect at Tivoli. He is working on Tivoli orchestration and provisioning technologies with a focus on metering and service provisioning.