SoLOMon: Monitoring End-User Service Levels

Svend Frølund
Hewlett-Packard Laboratories, frolund@hpl.hp.com

Mudita Jain
Hewlett-Packard Laboratories, jainm@hpl.hp.com

Jim Pruyne
Hewlett-Packard Laboratories, pruyne@hpl.hp.com

The first step in enabling the management of distributed applications to end-user expectations is the accurate and timely monitoring of end-user metrics. Monitoring distributed applications encompasses a large and rapidly changing set of users, applications, types of measurements, and platforms. This heterogeneity and large scale imposes two key challenges in monitoring end-user metrics: expressiveness and scalability. We need to express the metrics and their scope (both in time and space). The large scale, diversity, and dynamic nature of the scope of the measurements make it hard to specify metrics in a manner such that they are easy to comprehend, extensible, and uniform. These same reasons also make it hard to gather the described metrics in a scalable and timely manner. In this work we present the Activity Monitoring Language (AML), for declaratively specifying metrics, and a run-time system that implements the concepts in AML. Together, they compose SoLOMon (Service Level Objective Monitor), a system for the dynamic specification and monitoring of end-user service levels and events relating to (distributed) applications. SoLOMon is expressive, scalable, and extensible. Expressiveness in SoLOMon is a result of AML, which makes the system programmable. SoLOMon's scalability is a result of reducing events and measurements as close to their physical source as is possible without the loss of accuracy. The instrumentation independent nature of SoLOMon makes it extensible.

1. Introduction

The technology for monitoring enterprise-scale systems has not kept pace with the deployment of large scale, distributed, heterogeneous applications. This has introduced several challenges for the IT departments of enterprises. The set of users, applications, and platforms that they must manage is becoming very large and diverse, and is prone to rapid change. This environment makes it very hard to express uniformly and in an application independent way, the metrics that are to be managed, and to keep up with the changes in the scope of these metrics. Consider a usual task that a system administrator may have to perform. She might like to determine, for a particular SAP [1] application (say SAPclient_app) if the users in a particular workgroup see an average response time of 50 milliseconds or less. With traditional performance monitors, she would likely have to manually map "users in lab" to particular processes on particular machines. For example, she may have to state that she is interested in all machines on the subnet 15.25.57, if the user on the machine is in the group {joe, fred, ...}, and the machine is running the process SAPclient_app.exe. Moreover, once she has mapped "users in lab" to particular machines, she would likely have to manually extract the performance data from log files. For example, she may have to invoke a script on each machine to compute the average of the numbers in column 5 in the file "/usr/sap/data," and then aggregate the data across machines. The problems encountered by the system administrator in monitoring SAPclient_app can be summarized as follows:

- There is a large semantic gap between the specification of what she would like to measure, and the type of measurement that is actually available.
- Each application presents different types of measurements and different measurement interfaces to the user. Thus, each monitoring solution is application specific. There is no uniform way of carrying over the solution for monitoring one application to another, or of correlating metrics across applications.
- There is no easy, ubiquitous, scalable mechanism for gathering, correlating, and transporting the data from distributed sites to a central location.
- Most measurement systems do not have access to end-to-end metrics, and thus do not provide complete information.

The necessity of solving this problem is manifest in the number of products being released that aim at different subsections of the same space: "ApplicationExpert" from Optimal, "NETSYS Advisor" from Cisco, "VitalAnalysis" from VitalSigns, and "WebTrends for Firewall and VPNs" from WebTrends.

Traditional network and systems management products that are trying to move towards application management solutions of the type described above include HP (OpenView), IBM (Tivoli TME 10), Sun (Solstice), and CA (Unicenter). However, these "enterprise management systems" are extremely complex and expensive [7]. In fact, unable to deal with these management systems on their own, several companies are starting to out-source network and system management.

1.1 SoLOMon

In this paper we present SoLOMon (Service Level Objective Monitor), a system for the dynamic specification and monitoring of end-user metrics and events relating to (distributed) applications. SoLOMon is scalable, extensible, and instrumentation independent. The architecture of SoLOMon is illustrated in Figure 1.

The frontend is an authoring tool for the Activity Monitoring Language (AML). AML is used to tell the gateway component what to monitor. AML is best thought of as a programming language for a distributed measurement system. It is used to declaratively specify *what* to measure, without worrying about *how* to measure it.

The gateway component converts AML specifications into streams of measurements. Streams of measurements may have several consumers: graphical measurement visualizers, report generators, notifiers that apply thresholds to streams and produce events and alarms, and monitors that feed streams of measurements to other tools. These consumers may be implemented by anybody, and are not part of SoLOMon. In figure 1, we depict a monitor, reporter, and visualizer. These perform measurement control on a gateway component. The control operations specify which measurements the tools are interested in. In return, the gateway provides streams of measurements for the frontend tools.

The gateway component itself can receive measurements from two sources: agents in a measured system or a measurement repository. The measurement repository contains historical data whereas the agents provide current data. Both the agents and repository are subject to measurement control. The measurement control for agents turn low-level data providers on and off, and determine how measurements are aggregated. Providers represent the instrumentation points in the application being managed. Collectors allow hierarchical aggregation of data from providers. The provider abstraction gives a uniform interface to the heterogeneous instrumentation points in the various applications managed by SoLOMon.

The measurement control for the repository control aggregation and retrieval of historical data. The repository has a database in which it stores historical measurement data. We populate the repository through the monitor tool. The monitor tool obtains a live measurement feed from the gateway component based on AML specifications, and directs this live feed into the repository. The repository only contains the measurement that the user explicitly redirects to it—we can only present historical views on measurements that were explicitly collected.

In Figure 1 we use the term *measurement* to refer to measurement data at different levels of abstraction and reduction. The measurements coming from the agents in the network are likely to be at a lower level of reduction than the measurements being fed to the monitor tool—the measurements from agents are per-agent views, whereas the measurements being passed on to the monitor tool are likely aggregated across agents to provide a combined view.

We designed SoLOMon with the following aims.

- Provide abstractions that make it easier to map end-user business metrics to monitoring instructions. The instructions for traditional monitoring systems, such as Measureware and Glance [9], are more for resource monitoring rather than service monitoring.
- Provide scalable end-user monitoring. In monitoring true end-to-end user activity, we can expect to have a very large number of measurement points. It is essential that we can collect large-scale measurements without flooding the monitored system with collection traffic. Our approach is to perform measurement reduction as close to the measurement source as possible.
- Provide extensibility via a means for describing user-defined metrics and a means for integrating new types of instrumentation points while the system is in operation.

To provide notational convenience AML specifications were designed to be composable. Therefore it is possible to write complex specifications as a composition of a number of simple specifications. For example, it is be possible to construct the AML specification for an organization as a composition of specifications for the different entities that make up the organization. This notational convenience is important as we expect end users to visually compose and refine existing AML specifications rather than starting from scratch. In particular, we envision the frontend tools to come with a library of common AML specifications that can be tailored for the domain of interest to the end user.

The main contribution of AML is not its syntax, but the abstraction and formalization of the concepts that arise in the consideration of distributed monitoring.

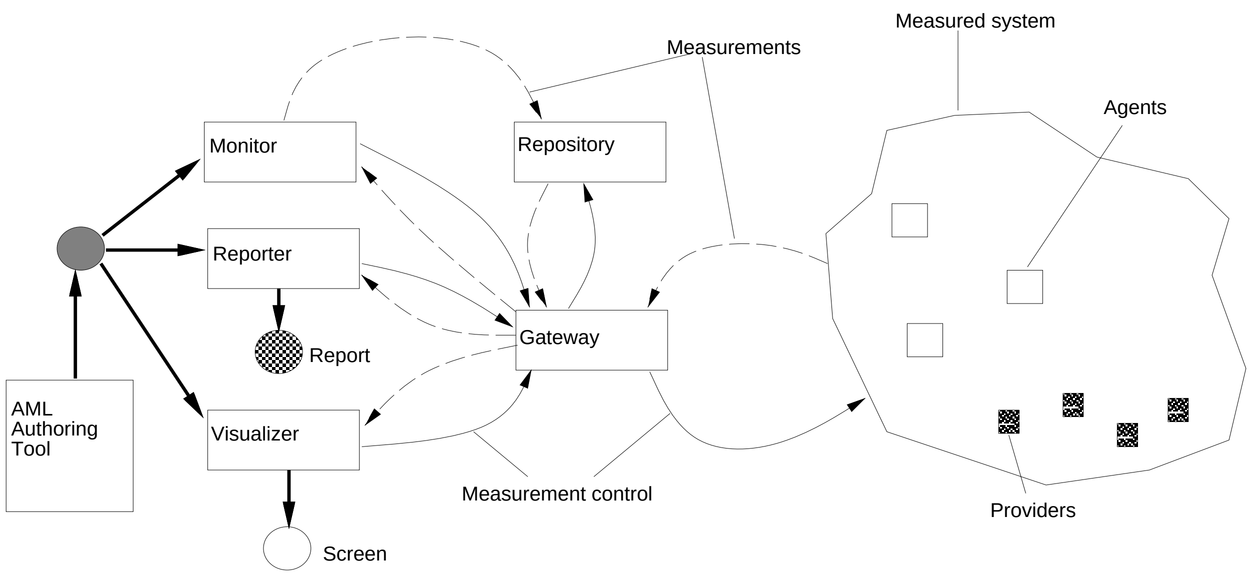

FIG. 1. Architecture of The SoLOMon Monitoring System

AML has a runtime representation that allows AML specifications to be first-class objects in the SoLOMon runtime system. We need this runtime representation so that the frontend tools can communicate AML specifications to the gateway component, and so that the gateway component can control the physical measurement providers.

We believe that the SoLOMon architecture can be leveraged by outsourcers. The SoLOMon concepts can help manage out-sourced systems according to service level agreements (SLAs). The key component of an SLA are the service level objectives (SLOs) that define the measurable performance goals of a system. An AML specification formally captures the concepts that comprise an SLO in a form that can be used to instruct a distributed measurement system.

SoLOMon also provides a framework that integrates the areas of event management and performance management. AML specifications can be written to perform much of the operational management functions performed by event management systems as well as producing the types of metrics associated with performance or service management. SoLOMon integrates the two by using a single infrastructure for both purposes, and using the same stream of observed events for both purposes.

The remainder of the paper is organized as follows. Section 2 describes AML in more detail, and section 3 describes an approach for implementing SoLOMon. Section 4 describes work related to SoLOMon. Conclusions and future work are presented in section 5.

2. AML: A Language to Program Distributed Measurement Systems

In this section we describe the design of the Activity Monitoring Language (AML), that provides a high-level frontend to distributed measurement systems. We use AML to declaratively specify *what* to measure (not *how* to measure it). The constructs of AML are designed with scalability in mind; we want to perform as much measurement reduction as close to the physical measurement source as possible. In AML, this goal manifests itself in constructs that can be sub-divided and therefore calculated in a distributed manner. In the next section, we describe these constructs, and how they combine to create high level views on a distributed system with simple instrumentation points.

2.1 Language Constructs

Each construct in AML is designed to perform one step in the process of transforming instrumentation or measurement into metrics with more semantic value. At the lowest level, we define *providers* and *provider types* which encapsulate different instrumentation or measurement points in a system. Above these, we define *metrics* which are procedures to convert these instrumentation points into values. Response time is a typical metric. The combination of a metric and a provider creates a *source* which can be viewed as a stream of values that can be controlled (e.g. turned on or off). The highest level construct is the *reducer* which transforms one stream of values (such as a source) into another. For example, a reducer may periodically output the mean value of a stream of incoming values. In the following sections, we describe each of these constructs in more detail by showing examples of how they are used.

2.1.1 Providers and Provider Types Provider types define the world of instrumentation in the system under consideration. Provider types are user-defined because the type of information (events) may vary widely from one domain to another, and new types are likely to arise in the future. For our monitoring infrastructure to work with heterogeneous information sources, both old and new, we need to provide a facility for describing the type of information available from any given source of instrumentation.

In Figure 2, we give an example of the AML syntax for defining a provider type for ARM [12] providers. ARM is an application instrumentation API, that provides a method of demarking the beginning and ending of a logical transaction. The definition has two sections: an attribute section and an event section. The attributes define properties of the providers of type ARM. In this case, providers have attributes that capture the user of the process in which the instrumentation is embedded, the name of the application, and the name of the activity (transaction) initiated by the enclosing process. Individual providers will bind names to these attributes. For example, if the process is owned by a user called "joe," the attribute user would be bound to the text string "joe." The binding of values to these names will happen at run-time, so will not be seen as part of the AML specification which is put in place a-priori.

The event section declares the different kinds of events that ARM providers can produce. In this case, we assume that ARM providers can produce `start` and `stop` events. These correspond to the API calls of the same names. Events also have attributes. The attributes of an event characterize it and help distinguish between different instances of the same event. For example, for `start` events, we want to know when the event occurred. Thus, `timestamp` is an attribute of the `start` event. An instance of a `start` event contains a value for the `timestamp` attribute. This value is of type `long`, and represents the time at which the `start` event occurred. As we see in the next section, event attributes also provide names used when defining metric procedures.

```
provider ARM {
  attributes
    user: string;
    application: string;
    activity: string;
  events
    start(tranID: uuid, timestamp: long);
    stop(tranID: uuid, timestamp: long,
      status int);
};
```

FIG. 2. The AML definition for an ARM provider type.

To further illustrate the notion of provider type, we give the AML definition for a hypothetical provider type that captures CCMS providers in Figure 3. CCMS is the name of the monitoring system in the SAP/R3 environment. In this example, a CCMS provider produces events called `responseTime`. These events correspond to the completion of a business transaction. In contrast to ARM events, where we get an event for both the beginning and end of a transaction, we get CCMS events only for transaction completions. If we want to compute a response time metric over ARM events, we have to compute the time difference between the `start` and `stop` events. With CCMS, the response time is already available as an attribute of the `responseTime` events.

2.1.2 Metrics The next higher level of abstraction in AML is the *metric*. A metric is a procedure that computes values from provider events. Figure 4 defines a metric to compute response time values. First, we use the keyword **metric** to define the type and unit of the metric. Then we define the procedures for computing values from events. We need to define a different procedure for each provider type as different provider types support different events. In this way, higher level constructs can be written in terms of metrics without regard to the underlying providers that generate the values. The definition in Figure 4 contains procedures for providers of type ARM and CCMS. The values computed by the response time metric are of type **float**. In general, metric values can be composite entities, such as pairs or records over other values.

The procedure to compute response time values from ARM events is complex because these values are computed based on two correlated events. We need the `start` and `stop` events from the *same* transaction to compute a valid response time. We therefore need to correlate events over a transaction. We use pattern matching over event attributes to describe event correlation. A pattern specifies acceptable values for (some of) the attributes in an event. For example, in Figure 4, we

```
provider CCMS {
  attributes
    user: string;
    application: string;
  events
    responseTime(eventID: uuid, time: long);
};
```

FIG. 3. The AML definition for a hypothetical CCMS provider type.

describe a pattern that matches all `stop` events whose attribute status has a value of "ok":

```
stop(status == "ok")                              (1)
```

Notice that in (1) we do not specify values for all attributes. If a pattern does not specify a value for an attribute, it trivially matches the attribute.

For the purposes of event correlation we need to describe patterns whose attribute values depend on attribute values in other events. For example, to correlate ARM `start` and `stop` events, we need to construct a pattern for the `stop` event that contains the `tranID` value from the `start` event. We use the following syntax to express this:

```
stop(status == "ok",tranID == t).timestamp -
  start(t = tranID).timestamp                     (2)
```

In (2), the `==` is a comparison operator and the `=` establishes a binding. The expression `t = tranID` establishes a binding for `t`. We can then use `t` in specifying patterns. Thus, when we write `tranID == t` we construct a pattern based on the value of `t`.

Notice that expression (2) may be evaluated only when all patterns are matched. Here there is only one pattern: the pattern for `stop` events. This pattern is matched whenever there is a successful `stop` event that has a `tranID` equal to that of a previous `start` event. We can access the attributes of the resulting matched event. To use the value of event attributes we use a dot (".") notation as follows:

```
stop(status == "ok",tranID == t).timestamp (3)
```

The result of evaluating (3) is the value bound to the attribute `timestamp` of the `stop` event that matches the contained pattern. We can then perform arithmetic expressions over these event attribute values. In the `response time` metric our operator is subtraction.

The computation of metric expressions over multiple events requires that events be stored. When a new event occurs, it may trigger the evaluation of a metric expression. For example, the occurrence of a `stop` event may cause the metric `response time` to be evaluated if a corresponding `start` event has occurred previously. In order to determine that such a `start` has in fact happened, we need to store old `start` events. One question then is: when can we safely delete events? We choose the semantics that the evaluation of a metric expression causes deletion of all the participating events. In other words, a given metric expression may use an event in only one evaluation. As measurement systems typically operate in resource constrained environments, forcing this semantics in the specification allows for efficient implementations. Although, events are removed after they have been used in a computation, we still need a policy to deal with events that are never used in computations. For example, we may never see a stop event that matches a given start event. The runtime system must ensure that we do not store such start events forever. One possible policy is to have an expiration time for events. Notice that garbage collection of events is a general problem for event-based measurement systems with correlation. For example, a measurement system based solely on ARM instrumentation would face the same issue.

```
metric responseTime: float msec;

responseTime for ARM {
  val =
    stop(status == "ok",
      tranID == t).timestamp -
    start(t = tranID).timestamp;
};

responseTime for CCMS {
  val = responseTime.time;
};
```

FIG. 4. Procedures that compute the response time metric for providers of type ARM and CCMS.

Consider the expression for computing response time values for CCMS providers:

```
val = responseTime.time;                          (4)
```

There are no patterns involved in expression (4). So we can evaluate this expression for each `responseTime` event. The result of the expression is the value bound to the event attribute called `time`. The `response time` metric definitions for ARM and CCMS provide us with a single notion of response time for both CCMS and ARM though the underlying events produced by the corresponding instrumentation are very different.

2.1.3 Sources and Filters Providers and metrics provide a definition of where events come from and how they combine to produce values. Here, we describe sources which represent a stream of values being produced by a metric. A source can be turned on and off at run-time. When a source is turned on, it produces a stream of values: the values computed from the metric over provider events. For example, we could create an ARM source that associates the `response time` metric with the providers that all support the ARM provider type. If we turned on the source, we would get a stream of `response time` values. Each value in the stream would correspond to two events: a `start` and `stop` event, at the underlying providers. We associate metrics with providers based on filters over the provider attributes.

The following is the definition of a source:

```
source responseTimeSource: float;
responseTimeSource = responseTime from
  filter { user == ``joe''};                    (5)
```

In (5), the first statement declares a source instance, `responseTimeSource`. `responseTime` is the name of a metric. The expression "**filter** { `user == ``joe''`}" instantiates a filter that matches on an attribute `user` in providers. The filter will select all providers where this attribute is bound to the text string "joe." The **from** keyword associates the filter with the metric. The second statement thus binds the source `responseTimeSource` to the result of associating the `responseTime` metric with all providers that have a `user` named "joe."

2.1.4 Reducers The association of a filter and a metric gives rise to a source. This is a primitive source, its values are computed directly by the metric expression. AML treats these sources as first class entities, and defines operators over sources so that new sources can be constructed from old sources. A reducer is an operator over sources.

One class of reducers provide time intervalization of measurement data. For example, **sum** is a built in reducer that takes a source S and a number T, and returns a source S'. Each value in S' is a sum of values from S. A sum value that equals the sum of S values over the interval T appears in S' every T time units. If over a particular interval no values appear in S, a special null value appears in S' as the value for that particular interval.

As illustrated in Figure 5, we can use the built-in **sum** reducer and a built-in **count** reducer to construct a user-defined reducer **mean**. The **count** reducer returns a source that contains values equaling the number of values from the input source over a given time interval. The `mean` reducer computes the time-averaged mean for the values in a source `S` given as input. The `mean` reducer also takes a number `period`, which represents the interval for time averaging. `mean` returns a source that contains the mean of `S` values computed on `period` time boundaries.

In Figure 5, we compute arithmetic expressions over sources. We use a division operator "/" on the sources returned by **sum** and **count**. The semantics of applying division to two sources is that the two sources are merged to provide another source. In general, the result of merging two sources according to a binary operator is a source whose values are constructed from a per-value application of the operator on the values of the two merged sources.

To describe the semantics of source operations, we need some way to operationally capture the semantics of a source. We use a stream of values to represent the runtime behavior of a source. We use the following notation to represent sources:

```
reducer mean(source S: float, period: long) {
  val = sum(S,period) / count(S,period);
};
```

FIG. 5. A reducer that computes the time-averaged mean value of the values from source `S`.

```
s = (t,v1 v2 v3 ...)                          (6)
```

(6) represents a source `s` that is time intervalized on an interval of length `t`. The source produces the values `v1`, `v2`, `v3`, and so on one after each interval `t`. With this notation, we can now capture the semantics of a binary source operation, such as division, in this way:

```
(t,v1 v2 v3 ...) / (t,v4 v5 v6 ...) =
  (t,v1/v4 v2/v5 v3/v6 ...)
```

In general, we only allow operators to work on sources that are either all unintervalized or that are intervalized with the same period. For unintervalized sources, the division operator would work as follows:

```
(,v1 v2 v3 ...) / (,v4 v5 v6 ...) =
  (,v1/v4 v2/v5 v3/v6 ...)
```

We allow the usual set of arithmetic operators over sources. In addition, we provide a number of built-in reducers to provide intervalization of sources. At this time, the built-in reducers are **sum**, **count**, **min**, and **max**.

Time intervalization of data requires state in the reducer that performs the time intervalization. For example, the **sum** reducer must store a sum value that is updated whenever the input source produces a value. The sum value is set to 0 on interval boundaries, and it is written to the output source before it is set to 0. It is hard, in general, to efficiently implement user-defined reducers with state in a distributed manner. We have therefore decided that user-defined reducers cannot have state. In particular, it is only built-in reducers that can accomplish time intervalization.

Distributed events may be combined into one value, and we do not assume clock synchronization or bounded clock skew between machines. Thus a value may be the result of two events that ostensibly occur at very different times. The semantics of a time-stamp for this value is unclear. The values produced by metrics and reducers therefore do not have an absolute timestamp as an implicit attribute. A result is that if we store unintervalized data in the repository, we cannot later intervalize this data. Similarly, when values are time intervalized, we ignore the time at which the value was itself computed. For example, due to transportation delays, a value may arrive at a reducer much later than when it was actually computed. Thus we choose to time intervalize a stream of values according to the clock at the reducer's computer. However, if we store intervalized data, we can increase the intervalization period when we retrieve it.

The `mean` reducer in Figure 5 provides a general way to describe time averaging of measurement data. For example, we can use the `mean` reducer to measure the average response time for the user "joe" in the following way:

```
interval = ...;
source responseTimeSource: float;
responseTimeSource = responseTime
  from filter { user == ``joe''};
avgResponseTime =
  mean(responseTimeSource,interval);
```

We use AML to declaratively construct source values that represent what we want to measure. AML describes the construction of sources, not the instantiation of sources. Source instantiation is done by the gateway component. The gateway provides a programming interface for turning sources on and off at runtime. For example, it might provide a method `instantiate` that given a source returns a stream of values, where the values in the stream are produced according to the source specification. The gateway might also provide a method to deactivate a stream. It serves the role of a stream factory, the entity that creates streams based on sources. We describe the interface between AML and the gateway in more detail in Section 3.

2.2 Discussion

We provide an extensible infrastructure that can handle information from a variety of heterogeneous sources, and reduce the information in a consistent manner across different sources. This extensibility and heterogeneity is not available in traditional measurement systems, such as DMS [16, 13]. The key to this extensibility is our notion of provider type that creates a common vocabulary for measurement sources. As we have previously mentioned, the concept of provider type serves many of the same purposes as interface definition languages, such as CORBA IDL. Where IDL provide language neutrality, provider types provide instrumentation infrastructure neutrality.

We use the concept of an event as the common denominator for heterogeneous measurement data. Producing events imposes minimal requirements on participating instrumentation points because the concept of an event does not require any processing, such as time intervalization, of measurement data. Having events at the base layer gives rise to a "push" model for measurement collection and processing. Some legacy systems are likely to support a "pull" model instead. We can integrate such legacy systems into the AML world by wrapping them by code that periodically pulls the legacy measurement system and produces an AML event. In the future we may consider allowing pre-intervalized data to be an external measurement source in AML.

One important question is where reducers can be computed. Our goal is to compute reductions as close to the measurement source as possible. However, distributing the computation of reductions is not always possible. It turns out that we can (statically) identify the class of distributable reducers.

Consider the expression $t = f(s_1, \ldots, s_n)$, where t and $s_1 \ldots s_n$ are sources. The function f represents the expression over the sources $s_1 \ldots s_n$. If the sources are unintervalized, we compute a value for t when all sources s_i have a value. We compute one value for t based on n values—one from each s_i. If f is a built-in reducer, or if all sources s_i are identical, we can distribute it, otherwise it is computed by the gateway. If the s_i are intervalized on the same interval, the source t will have that interval as well. At the end of each interval, the n values from the sources s_i are combined into one value for the source t.

3. Implementing AML

The implementation of AML needs to satisfy the following properties.

- It must be scalable. That is, it must support large numbers of data sources on a large number of distributed resources.
- It must be low overhead. This requirement not only supports the need for scalability, but it also ensures that SoLOMon does not overly perturb systems on which it is running.
- It must be timely. The goal of SoLOMon is to provide nearly real-time monitoring of a distributed system. The implementation must, therefore, report results soon enough after the events that compose them occur.
- It must be extensible. Over time, new primitives or extensions to the base specification language may be made. The implementation needs to support these updates without requiring significant changes. In addition, the language itself may define encapsulations for new data sources, and it should be easy to integrate these data sources with the implementation.

The implementation is encapsulated in an "agent", which is simply a process or daemon that runs on every machine on which AML specifications can be executed.

3.1 Dataflow Expression Trees

Both reducers and metrics contain expressions. Metric expressions compute values from events and reducer expressions combine sources. Both types of expressions are represented as dataflow expression trees in the runtime system. An expression tree represents a programmatic statement as a collection of nodes that represents specific operations, and link to sub-nodes that represent the operands. For an introduction to dataflow, refer to any of [6, 2, 8].

Traditionally, the evaluation of an expression tree is a simple recursive descent of the tree in which the child operation nodes are first evaluated, and their values are used to evaluate the operation node at the current level. However, AML expression trees are evaluated in a bottom-up manner. For example, an expression tree that represents a metric will have leaf nodes that corresponds to provider events. The tree evaluation is triggered by event occurrences at the leaf nodes. The expression tree for a reducer will also be evaluated bottom-up. Here, the leaf nodes represent the sources being reduced, and the root represents the source which is the result of the reducer. The evaluation of a reducer expression tree is triggered by values being produced by the leaf sources. Due to their asynchronous nature, we refer to the runtime expression trees as dataflow expression trees.

The expression of each individual metric and reducer will be represented as a tree. However, these trees will be composed at runtime, reflecting the instantiation of sources. Thus, the representation of an instantiated source is, in general, a dataflow graph that is built from dataflow expression trees.

In the following, we describe the translation of each of AML's constructs into expression trees. As an example, we will use the AML metric defined in Figure 6.

This is a simple metric that takes the response time from a CCMS sensor, and multiplies its value by two.

3.2 Implementing Sources

As suggested by the AML syntax, a "provider" is a source for generating events. When a provider registers with an AML agent, it informs the agent about its attributes, the values bound to those attributes, and the events that it may generate. The AML agent is able to accept inputs from a variety of sources, as long as they conform to the "provider" Application Programming Interface (API). In some cases, such as ARM [12], an existing event-based API can be easily implemented as a pass through to the AML agent. In other cases, translation code layers can be used to pass events or measurements taken from other systems into the AML agent. For example, a management system such as WBEM [4] gives access to a great number of system metrics. To provide the ability to monitor WBEM information, the site-IT organization using AML would create a translation layer between WBEM and AML. This translation layer implements the "provider" language construct of AML. As mentioned previously, we expect the AML compiler to generate stub code from provider type specifications. The stubs generated for a particular provider type will define the data format for registration and event generation for providers of that type.

When an event occurs, the provider generates an event object, populates it with values for each of its fields, and passes it into an "eventInput" node via the agent's API. As illustrated in Figure 7, the "eventInput" node is the means of putting events into the expression tree context. These nodes form the interface between the AML evaluator agent and the provider. In practice, this interface is presented as a specific API. This API contains methods for registering providers when they start up and for transporting events from these providers into the AML evaluator agent. This API needs to be highly optimized to reduce the overhead associated with the generation of events.

The following are some of the issues that arise in implementing the provider abstraction and the Event Input node.

- Filtering. Recall that AML supports the ability to filter incoming events according to their attributes. Hence, if an event has the attribute user, then we can filter for response times for user "joe". In implementing the provider abstraction, we need the ability to execute these filters, which will be handed down in executable form by the run-time support. Evaluating these filters at the source is crucial to the scalable nature of AML's implementation.
- Event distribution. There may be more than one high-level goal that is subscribing to a particular event. Assuming that there is an expression tree per goal, the event needs to be distributed to the Event Input nodes of each expression tree that subscribes to it.
- Garbage collection. We need to know when an event is no longer useful and can be safely discarded. This knowledge is best kept in the "eventInput" node, and not the provider abstraction. A variety of strategies for garbage collecting events are possible including bounding the total buffer size, or the lifetime of a single event.

```
responseTimeX2 for CCMS {
  val = responseTime.time * 2;
};
```

FIG. 6. A simple metric

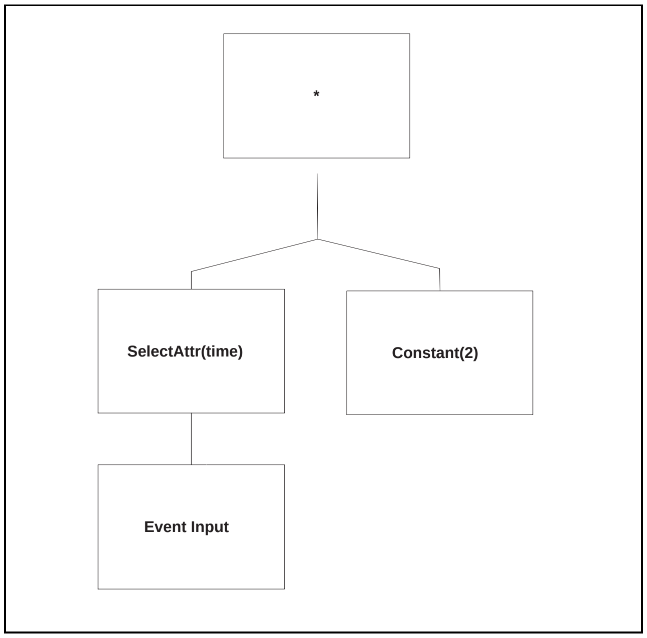

FIG. 7. The expression tree for the simple metric

3.3 Implementing Metrics

Figure 7 shows the expression tree that represents the metric in Figure 6. At the bottom, the node labeled `EventInput` corresponds to an event entering the system. This node simply propagates the event up the tree to it's parent the `SelectAttr` node. This node selects a specific event attribute value, in this case the `time` attribute, and propagates it up the tree to the multiplication node. When the multiplication node receives the value, it can look to its children to determine if they all have values ready. In this case they will because the constant node always has a ready value. When the multiplication node is evaluated, it will pass its value on up to its parent.

This approach is simple, and it helps satisfy a number of our criteria, including extensibility and timeliness. We can extend AML with new language features by introducing new types of tree nodes without breaking the overall structure. As an example, Section 3.5 describes how this approach permits us to implement AML in a distributed environment. Additionally, in some dynamic environments such as Java or ActiveX, we can load the implementations for new types of nodes into a running implementation. Because the evaluation is driven by event occurrences rather than elapsed time, the results generated will be timely with respect to the events that generate them.

3.4 Implementing Reducers

As discussed previously, the AML language introduces a set of built-in reduction operations: **sum**, **count**, **max**, and **min**. These reduction operators are commutative and associative. Thus, they have the important property that we can perform these operations on a subset of all values, and combine these intermediate values to determine the proper global value. That is,

```
a + b + c = (a + b) + c =
    a + (b + c) = (b + a) + c
```

These properties permit us to perform reductions in a distributed fashion, and bring the results together at a later time to get the global result.

To implement reducers in AML, we introduce new expression tree nodes corresponding to each of the built-in reducers. Unlike normal expression tree nodes, these nodes accumulate their values over an interval (note that in the AML specification, these reducers require a period over which to reduce), and pass their values up the tree only after that time has elapsed. At the beginning of each interval, the current value is set to a null value so that accumulation can begin fresh for the next interval. By accumulating at the node, we need not change the behavior of the expression tree evaluator or the behavior of other nodes. Note that we do not consider the issue of untimely delivery of events. The reducer accumulates all values rceived during a time interval regardless of when these values were actually computed at their physical source.

3.5 Distributed evaluation

Evaluating AML specifications in a distributed environment presents some interesting problems. First, how do we accurately evaluate expressions involving multiple machines. Second, what is the mechanism by which distribution is performed. In both cases, the extensibility and flexibility of the tree structure is the key.

Like reduction, distribution relies on our ability to introduce new expression tree nodes to create new functionality. To perform distribution, we simply introduce "communication" nodes into the tree. When one of these nodes receives a value, it communicates it to another communication node on another machine that receives the value, then passes it up its expression tree. Different forms of communication nodes can be implemented to handle different communication infrastructures. In this way, we can easily port the entire AML evaluator to a new communication subsystem by implementing only one type of node. The rest of the system is insulated from these changes.

Other issues in distributed evaluation include timeliness, naming issues (domains), and scalability (evaluating attributes when they are encountered, filtering).

Figure 8 combines these concepts by showing how the mean reducer of Figure 5 would be handled in a distributed environment. At the bottom, we assume a value has been created, perhaps by the multiplication node of the previous example. This node feeds both the `Sum` and the `Count` nodes locally. This corresponds to the fact that a source, S, in the reducer definition is passed to both the **sum** and the **count** reducers. Both the `Sum` and `Count` nodes are parameterized by the interval, so they accumulate all of their values during that interval, then pass them up to their corresponding communication nodes. These nodes simply pass the value across the communication substrate to the corresponding communication nodes on the other end. The subtrees below the lowest communication nodes would be replicated across all hosts in the system. The upper communication nodes would only be on a central node, and would pass values they receive up to the two `Sum` nodes. Note here that the global operator to reduce individual `Count` operations is a sum, not another count. Finally, when the interval is complete, the two values are passed up to the division operator where the final, global mean is computed. Because of the semantics of a mean, the only place possible to do this division is at a global point where all values are known.

4. Related Work

HP's Data Source Integration (DSI) language [11] is used in conjunction with the HP Measureware collection agent [9]. DSI allows external measurement providers to use the Measureware collection infrastructure. A DSI specification defines the format of a particular class of measurement data that is produced by an external measurement provider. In addition to the data format, DSI also specifies what the measureware agent should do with the data. For example, a DSI specification may

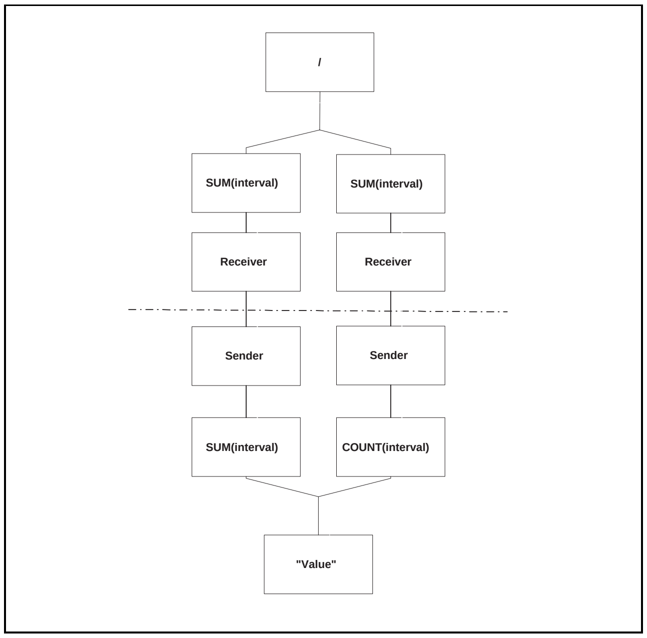

FIG. 8. An expression tree computing a mean across two systems

contain instructions about logging policies and intervalization periods. In contrast to AML, DSI only addresses the interaction between measurement providers and a single collection agent. DSI does not support distributed and hierarchical measurement reduction, not does it support control of measurement providers based on their attributes.

The InfoVista system [5] contains a language for describing metrics. It is possible to describe composite metrics in terms of simpler metrics. However, InfoVista does not support distributed computation of metrics: composite metrics are computed at a centralized measurement server. Instead of attribute-based naming of measurement sources, InfoVista uses a model-based approach. Each managed element (in our case each application component with an instrumentation point) is represented as a separate object in InfoVista. Measurement control is then expressed as explicit binding of metrics to managed elements. We believe attribute-based naming is more scalable, and therefore more appropriate for monitoring end-user metrics. The InfoVista language does not support computation of metrics based on events. The base level of the language is primitive metrics, which are computed outside of the InfoVista system.

The distributed measurement system (DMS) [16, 13] is an architecture for collecting performance measurements from distributed applications. DMS has per-computer measurement agents that collect, intervalize, and transport measurement data to a central location. An agent can receive measurement data from multiple measurement sources on its computer. The notion of threshold in DMS allows control of individual measurement sources. They can be turned on and off and instructed to operate at different levels of data aggregation. A threshold corresponds to an intensity level for measurement data. However, DMS only provides a fixed set of intensity levels, there is no way to define custom thresholds and use them for existing DMS measurement sources. Moreover, DMS measurement sources have unique names, there is no support for attribute-based naming. Finally, DMS does not separate the concepts of metric and intervalization: intervalization is an inherent property of metrics. For example, there is no way to obtain unintervalized response time data.

The stream concept in dataflow languages, such as Lustre [14] and ESTEREL [3], is similar to our notion of a source. Where AML provides reducers to perform computation of sources, these languages provide operators to perform computation over stream values. However, the data operators in Lustre and ESTEREL do not support any notion of intervalization. Furthermore, Lustre and ESTEREL do not support attribute-based naming of data sources, nor do they provide operators to specify the computation of values based on events.

A number of management systems are based on the observation and correlation of events. The MODEL system [15] is an example of an event correlation system. HP's IT/Operations (IT/O) [10] is another example. Event correlation systems generally observe events at multiple locations in a distributed system, and attempt to reduce the low-level events observed to higher-level events that capture the state of the managed system. Where these systems convert measurement data and low-level events to higher-level events, AML converts low-level events to measurement data using metrics.

5. Conclusion

Managing large-scale distributed enterprise applications requires that we can accurately and efficiently monitor end-user metrics, such as availability and response time. Two key challenges in monitoring such metrics are scalability and expressiveness. The population of users is typically very large, and the monitoring system must be able to scale to very large numbers of instrumentation points. Also, the expression of measured metrics and the scope of measurement (both time and space) must be flexible to accommodate a wide range of management functions. We have introduced SoLOMon, a distributed monitoring framework designed to provide scalability and expressiveness. SoLOMon's scalability is primarily achieved by reducing measurements as close to their physical source as possible. Expressiveness in SoLOMon is a result of using a high-level language, AML, as a declarative front end to the measurement system, making the system programmable.

The SoLOMon framework is built around the notion of reducing events to values. A topic for future work would be to extend the framework to provide ways to generate events from values. Another extension is to provide operators that reduce events to other events.

6. Acknowledgements

The work reported in this paper has benefitted greatly from interactions and discussions with a number of individuals. We want to acknowledge the contributions of Muthusamy Chelliah who participated in the early design meetings. Joe Martinka continually supported and encouraged our endeavors into distributed measurement systems. We thank Joe Sventek for his comments on a draft version of this paper. Finally, we thank Brad Askins and Pankaj Garg for their feedback on our work.

References

1. SAP AG. *Database Administration*, chapter 9. SAP AG, May 1997.
2. "Arvind, K.P. Gostelow, and W. Plouffe". "an asynchronous programming language and computing machine". Technical Report "UCI-TR114a", "U.C. Irvine", "Dept. of Information and Computer Science, UC Irvine", "December" "1978".
3. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, and implementation. *Science of Computer Programming*, 19:87–152, 1992.
4. Industry Consortium. Web based enterprise management initiative. http://wbem.freerange.com, 1997.
5. InfoVista Corporation. It quality of service management solutions. Technology white paper from http://www.infovistacorp.com/, March 1997.
6. "J.B. Dennis". *"First Version of a Data Flow Procedure Language"*, volume "19" of *"Lecture Notes in Computer Science"*. "Sprintger-Verlag", "1974".
7. Caryn Gillooly. Enterprise management: Disillusionment. *Information Week*, pages 46–56, February 1998.
8. "J.R. Gurd, C.C. Kirkham, and I. Watson". "the manchester prototype dataflow computer". *"Communications of the ACM"*, "28"("1"):"34–52", "January" "1985".
9. Measureware agent: User's manual, 1995.
10. Hp openview it/operations concepts guide, June 1996.
11. Measureware agent: Data source integration guide, June 1996.
12. Hewlett-Packard. Application response management. http://www.hp.com/-openview/rpm/arm/index_f.html, 1997.
13. J.Martinka, R. Friedrich, and T. Sienknecht. Murky transparencies: Clarity through performance engineering. *Proc. of the Intl. Conf. on Open Distributed Processing (ICODP'95)*, February 1995.
14. N.Halbwachs, P.Caspi, P.Raymond, and D.Pilaud. The synchronous dataflow programming language lustre. *Proc. of the IEEE*, 79(9), September 1991.
15. D. Oshie, A. Mayer, S. Kliger, and S. Yemini. Event modeling with the model language. In A. Lazar, R. Saracco, and R. Stadler, editors, *Integrated Network Management*, 1997.
16. R.Friedrich, J.Martinka, T.Sienknecht, and S. Saunders. Integration of performance measurement and modeling for open distributed processing. *Proc. of the Intl. Conf. on Open Distributed Processing (ICODP'95)*, February 1995.