Event-based Multi-level Service Monitoring

Luciano Baresi and Sam Guinea
Politecnico di Milano
Via Golgi, 42 – 20133 Milano, Italy
{luciano.baresi | sam.guinea}@polimi.it

***Abstract*—Due to the growing pervasiveness of the service paradigm, modern systems are now often built as software as a service, and tend to exploit underlying platforms and virtualized resources also provided as services. Managing such systems requires that we be aware of the behaviors of all the different layers, and of the strong dependencies that exist between them. In this paper we present the Multi-layer Collection and Constraint Language ($mlCCL$). It allows us to define how to collect, aggregate, and analyze runtime data in a multi-layered system. We also present ECoWare, a framework for event correlation and aggregation that supports $mlCCL$, and provides a dashboard for on-line and off-line drill-down analyses of collected data. Empirical assessment shows that the impact of the approach on runtime performance is negligible.**

Keywords*—*Service-based systems, Cloud computing, Multi-level service monitoring, Service performance analysis

I. Introduction

The service paradigm, together with virtualization technology, is imposing a profound re-thinking of many complex software systems. Providing virtual infrastructures as services is gaining more and more momentum, and is imposing a more *cohesive* view of the different layers that constitute a software system. Applications, service platforms, and virtualized infrastructures have become tightly integrated. It is now possible to change a software's quality of service (QoS) by changing the configuration of the virtual machines it uses. We can even instantiate new virtual machines to address load problems, or move our software from one infrastructure to another if its quality is not acceptable. Even if one might say that this is nothing new, the key distinctive feature is the *ease* with which the different parameters can be changed, and the different runtime executors (e.g., virtual machines) added or modified to impact a system's behavior.

Originally, monitoring of service-oriented systems was only performed at the application layer (e.g., [1], [2], [3]), and the lower layers were considered to be a constant. Recently, however, cross-layer monitoring is imposing itself as a promising and challenging research problem. Available technologies provide users with means to tackle the problem of the quality of service of these applications by digging down into the different layers: [4], [5], [6] are some first proposals that try to address this problem in a more comprehensive way. However, the more complexity we want to tame, the more sophisticated our solutions must become. If we take advantage of the ease —and low impact— of installing software probes within the different layers, the amount of data we collect can grow very rapidly. This calls for efficient and precise methods for their management. We need a customizable and extensible way to collect, aggregate, and analyze data, in order to identify the causes of anomalous behaviors.

In this paper we introduce a new approach for the cross-layer monitoring of complex service-based systems. The approach proposes two main novel contributions. We present $mlCCL$, a novel and extensible declarative language that designers can use to define (i) the various data they want to collect from the layers in their system, (ii) how to aggregate them to build higher-level knowledge, and (iii) how to analyze them to discover undesired behaviors. We also present ECoWare (Event Correlation Middleware), an event correlation and aggregation middleware that supports $mlCCL$ specifications. It provides advanced data aggregation and analysis features, and can be used to probe systems that are based on the Service Component Architecture (SCA) and deployed to virtual resources. Ecoware also provides a dashboard for reasoning on multiple dimensions of a running system at the same time, and for performing drill-down analyses to discover the causes of a revealed anomaly.

The paper's contributions are exemplified in the context of a simple cloud-based application called SocialTools[1]. SocialTools consists of a set of services, built as SCA composites and hosted on Amazon EC2, for gathering rich up-to-date information about users of Social networks. It integrates location-based information, taken from Twitter tweets, with weather information, taken from WebserviceX.NET.

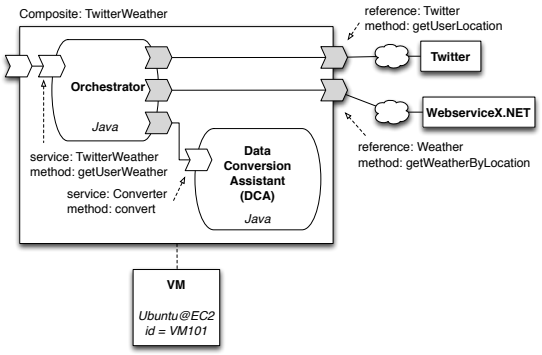

Fig. 1. The design of the SocialTools service. The Orchestrator SCA composite is shown as a rectangle. It contains two SCA components, shown to have rounded corners. SCA services and references are shown as white and light gray arrows respectively.

Figure 1 shows the design of our SocialTools application. We have a main orchestration SCA component called `Orchestrator`. It offers a service called `TwitterService`, which is also promoted and offered by

[1]SocialTools is an in-house extension of a SCA application made publicly available by Frascati (http://frascati.ow2.org), a well-known open-source implementation of the SCA standard.

the TwitterWeather composite. It requires three different partner services, as indicated by its three references. The first two point towards external services provided by Twitter and WebserviceX.NET; the third points to a local Data Conversion Assistant (DCA) that provides the data conversions needed to integrate Twitter and the Weather service. The entire composite is deployed to an Ubuntu VM installed on Amazon EC2 identified by id VM101.

The rest of the paper is organized as follows. Section II presents $mlCCL$, and explains how it can be used to collect, aggregate, and analyze runtime data. Section III introduces the ECoWare tooling provided to execute $mlCCL$ directives. Section IV illustrates the use of $mlCCL$ and ECoWare on two scenarios in the SocialTools example. Section V surveys related work, and Section VI concludes the paper.

II. $mlCCL$

Existing declarative event processing languages, such as Esper [7] and Tesla [8], are extremely powerful; yet, the fact that they are general-purpose also makes them quite complex. This is why we have decided to provide a new language, one with key domain-specific abstractions for multi-level service-based system. It is called the "Multi-layer Collection and Constraint Language" ($mlCCL$); it is easily extensible, and can be used to specify how to collect, aggregate, and analyze runtime data.

A. Data Collection

$mlCCL$ is based on the assumption that data are described in terms of Service Data Objects (SDOs). A Service Data Object is a language-agnostic data structure typically used to facilitate the communication between diverse service-based entities. A data object consists of a set of named properties. Each property can be either single- or multi-valued (i.e., an array), and can have either a primitive (i.e., a number, a string, or a boolean) or complex data-type (i.e., a service data object). This very simple assumption allows us to aggregate and analyze any kind of runtime data, as long as we are able to collect and correctly embed them into SDOs. $mlCCL$ allows for two different kinds of data collections: messages and indicators.

1) Message Collection: Message collection is used to obtain the request or response messages that are exchanged during service invocations. The first step is to determine the "location" where the collection will take place. This is done using the following syntax:

```
locName = [before | after] endpoint;
```

where locName is a location name chosen by the designer, before and after are $mlCCL$ keywords, and endpoint is a (methodName, [serviceName | referenceName], componentName) triple that uniquely identifies a method in a SCA-based system. If the triple refers to a SCA service, the messages are collected right before or right after methodName is invoked on componentName; if the triple refers to a SCA reference, the messages are collected right before or right after componentName calls methodName on its partner service. Message collection can therefore be defined as:

```
aliasName = collect(locName);
```

where aliasName is an alias that will be used to refer to the collected SDOs, and collect is a function that takes a location (locName) as its sole parameter.

Every time there is a new service invocation for which we have defined a message collection, the $mlCCL$ tool support produces a new SDO and outputs it to an event bus so that the designer can make further use of it. The SDO contains the collected message, as well as the aliasName and location values used in the collection specification. On top of that, it also contains a timestamp indicating "when" the message was sent or received by the service runtime[2], and an instanceID, that is a unique ID that identifies the specific service call. Different instanceIDs are generated for successive service calls, while the same instanceID is shared by corresponding request and response messages so that they can be easily correlated.

With reference to our running example, if we want to collect the twitter usernames that are being sent as inputs to our composite service, we need to collect the request messages that are sent to method getUserWeather, provided by service TwitterWeather, exposed by the Orchestrator SCA component. If we want to collect the users' geographical locations and the weather they are witnessing, we need to collect the response messages for the same service method. This can be achieved by writing:

```
locBefore = before(getUserWeather,
                TwitterWeather, Orchestrator);
locAfter = after(getUserWeather,
               TwitterWeather, Orchestrator);
request = collect(locBefore);
response = collect(locAfter);
```

2) Indicator Collection: Indicator collection is used to obtain periodic information about a service, and is not triggered by any particular service call. An indicator can be a Key Performance Indicator (KPI) of a software service, such as its average response time or its throughput, or a Resource Indicator (RI) of an infrastructure service, such as the amount of available memory or idle CPU in a virtual machine. This is defined using the following syntax:

```
aliasName = collect(indicatorName, serviceID,
               outputRate, pastWindow, property);
```

where aliasName is the alias that will be used to refer to the collected SDOs, and collect is an overloaded version of $mlCCL$'s collection function that takes five parameters. indicatorName is the name of the indicator we want to collect, while serviceID identifies the service for which we are requesting the indicator. In case we are collecting a KPI, the serviceID is an endpoint triple, like the ones defined for message collection but without the before or after keywords. In case we are collecting a RI, the serviceID is the unique ID of a virtual machine surrounded by brackets. The outputRate identifies the frequency with which a new

[2]In our $mlCCL$ runtime support we use NTP (Network Time Protocol) timestamps generated using Apache's Commons Net implementation. It guarantees a synchronized clock when sampling sources on different computers. Its precision is in the order of 10 milliseconds.

indicator value should be produced, while the pastWindow parameter specifies the amount of past time to take into consideration when calculating the new value. The pastWindow parameter is defined by a value and a time unit that can be "seconds", "minutes", or "hours", and can be set to null if the indicator does not depend on any past data. For example, if we want to know, every five minutes, what a service's average response time has been over the last hour, the outputRate would be "5 minutes", and the pastWindow would be "1 hour". Finally, property is a $mlCCL$ data analysis expression that is used by indicators that, in order to be calculated, need to know how many SDOs, collected in the last pastWindow, satisfy a given property (e.g., service reliability). The expression uses the data analysis syntax that we will illustrate in Section II-C.

Every time $mlCCL$'s runtime support calculates a new indicator value, it is wrapped in an SDO and output to an event bus so that the designer can make further use of it. The SDO contains a property called indicatorName with the name of the KPI or RI, and a property called value with the new indicator value. On top of that, it also contains the aliasName, serviceID, outputRate, pastWindow, and property values used in the collection specification. Finally, each SDO also contains a timestamp property indicating "when" the new value was produced.

Currently, $mlCCL$ supports four software service KPIs and fifteen infrastructure RIs. The four KPIs are avgRT, rate, throughput, and reliability. avgRT outputs the service's average response time. rate outputs the number of requests that were made to the service. If the designer specifies a property, the calculation of the indicator will only take into account the requests that satisfy that property. In this case the property can make use of an implicit alias called "input" to refer to the contents of each collected request message. throughput outputs the number of serviced requests, and reliability outputs the number of correct outputs over the total number of requests. The correctness of an output is determined by the fact that it satisfies a defined property. In this case the property can make use of implicit aliases "input" and "output" to refer to matching request and response messages. This is useful if the response message's correctness depends on the content of the corresponding request message. With reference to our running example, if we want to collect, every five minutes, the Orchestrator's arrival rate and average response time, calculated over the last hour, we need to specify:

```
orchestratorEndpoint = (getUserWeather,
                TwitterWeather, Orchestrator);
rate = collect(rate, orchestratorEndpoint,
          5 minutes, 1 hour, null);
avgrt = collect(avgRT, orchestratorEndpoint,
          5 minutes, 1 hour, null);
```

The fifteen infrastructure RIs are cpuIdle, cpuUser, and cpuSystem for CPU activity; diskOctets and diskOps for disk activity; memUsed, memBuffered, memCached, and memFree for memory activity; and netIncoming, netOutgoing, netPacketsTX, netPacketsRX, netErrorsTX, netErrorsRX for network activity. When collecting these RIs we do not need to specify any pastWindow and property parameters.

With reference to our running example, if we want to collect information, every five minutes, about the virtual machine's use of its CPU, memory, and incoming bandwidth, we need to write:

```
vm = (VM101);
cpu = collect(cpuSystem, vm, 5 minutes, null, null);
mem = collect(memUsed, vm, 5 minutes, null, null);
net = collect(netIncoming, vm, 5 minutes, null, null);
```

$mlCCL$ was designed so that it could be extended to be used with other service layers. This is achieved by adding new collectible indicators. Obviously, the addition of a new indicator will also require updates to the runtime support (see Section III-A), and in particular the addition of new kinds of probes.

B. Data Aggregation

Data aggregation is the operation that allows us to aggregate multiple SDOs into a single one. Since different SDOs are typically collected at different times, data aggregation needs to specify "when" the aggregation should take place, and "what" SDOs should be involved. In $mlCCL$, the "when" is determined by the collection of a "primary" SDO, which triggers the aggregation of a list of "secondary" SDOs. Data aggregation is specified using the following syntax:

```
aliasName = aggregate(primary, list);
```

where aliasName is the alias of the new aggregated SDO, primary is a previously defined alias, and list is a comma-separated list of previously defined aliases.

Every time a new SDO is collected for the primary parameter, the $mlCCL$ runtime aggregates the last known value of each of the secondary parameters. Since this can be limiting we also allow the designer to request a past window of SDOs for each of the secondary aliases. This is done by appending the window(interval) function to the desired alias, where the interval is defined using the same syntax used for the pastWindow parameter in indicator collection specifications.

The resulting aggregate SDO contains a property, of type SDO, for each of the aliases involved in the aggregation (both primary and secondary). On top of that, it contains a primary property with the name of the alias used for the primary collection, and a timestamp property indicating when the aggregate was created. Notice that if one of the secondary aliases uses the window function, its corresponding property will be multi-valued (i.e., an array of SDOs).

With reference to our running example, if we want to aggregate Twitter's average response time with its invocation rate in the last hour we can write:

```
avgrt_rate = aggregate(avgrt, rate.window(1 hour));
```

C. Data Analysis

Data analysis allows us to reason over the SDOs we collect and aggregate, by predicating over their internal primitive properties. In $mlCCL$ we can refer to a SDO's internal property by appending method get(propertyName) to its

alias. Depending on the type of property we extract, $mlCCL$ also provides a number of methods for further manipulation, such as absolute value and square root for numbers; and substring, length and replace for strings. If the SDO is an array, we can obtain the length of the array, the i-th value in the array, or a subset of all the values that satisfy a given property. In this case the property can refer to the elements in the array using a pre-defined `elem` alias. Finally, if the SDO is an array of numbers we can obtain the sum, the average, or the minimum or maximum values of the array.

With data analysis we define an expression that, when verified, causes a new SDO to be output to the event bus. Data Analysis is defined using the following syntax:

⟨prop⟩ ::= ¬⟨prop⟩ | ⟨prop⟩&&⟨prop⟩ | ⟨prop⟩‖⟨prop⟩ |
⟨array⟩ . ⟨quant⟩ (⟨prop⟩) | ⟨term⟩⟨rop⟩⟨term⟩
⟨term⟩ ::= ⟨prim⟩ | ⟨term⟩⟨aop⟩⟨term⟩ | ⟨const⟩ |
⟨rop⟩ ::= < | ≤ | == | ≥ | >
⟨quant⟩::= *forall* | *exists*
⟨aop⟩ ::= + | − | × | ÷ | %

where `prim` is a primitive value (i.e., number, string, or boolean) extracted from a SDO, and `array` is an array of primitive values, all of the same type. Boolean, relational (`rop`), and arithmetic operators (`aop`) follow their usual definitions. `forall` and `exists` state that a property should hold for all, or for at least one, of the values in an array. In this case the property can refer to the elements in the array using an implicit `$elem` alias. Data analysis poses the issue of understanding when the analysis should be performed. Obviously we need all the data to be available before we can proceed to evaluate an expression. Just like in data aggregation, we resort to the use of a "primary" SDO. The syntax for defining a data analysis is therefore:

```
aliasName = evaluate(primary, expression);
```

where `aliasName` is the alias that is used to identify the SDO that is created every time `expression` holds, while `primary` is a previously defined alias that is used within `expression`. When the primary alias is collected, the evaluation of the expression is triggered.

The SDO that is created contains an `alias` property with the alias name used in the data analysis specification, a `primary` property with the name of the primary alias, an `expression` property with the $mlCCL$ expression defined in the specification, and a `timestamp` property indicating "when" the SDO was created.

With reference to our running example, if we want to output a new SDO every time the average response time of method `getUserWeather` surpasses a 5500 milliseconds threshold we need to specify:

```
violation = evaluate(avgrt, avgrt.get(value) > 5500);
```

Since data analysis outputs new SDOs, these SDOs can also be used as primary aliases for other data aggregation or data analysis specifications. For example, if we want to aggregate the arrival rates collected in the last 30 minutes every time there is a violation in the average response time, we can write:

```
violation_rate = aggregate(violation,
                    rate.window(30 minutes));
```

III. The ECoWare Framework

ECoWare, which stands for Event Correlation Middleware, is a distributed event correlation and aggregation tool[3] that we initially developed for the monitoring of BPEL processes [9]. However, it is general purpose and can be used with different kinds of service technologies, as long as we provide appropriate probes.

For this paper we extended EcoWare to support SCA services and Ubuntu virtual machines, and provided new analysis capabilities, and an entirely new data visualization tool, to support our $mlCCL$-based analyses. An ECoWare deployment consists of three different types of components (see Figure 2): (i) the execution environments for which we want to collect run-time data (together with appropriate probes), (ii) processors for providing data aggregations and analyses, and (iii) the ECoWare Dashboard for visualizing collected data. The components collaborate through a Siena [10] Publish and Subscribe (P/S) event bus using a normalized message format.

The components that make up an EcoWare deployment depend on the $mlCCL$ specifications we provide. We have defined XML configuration templates for probes, data processors, and the EcoWare dashboard, so that they can be easily completed and deployed according to the specification's needs. During this configuration phase we also wire the ECoWare components through appropriate Siena publish and subscribe ids, so that, together, they can produce the information we require. The configurations are optimized to minimize the number of components that are actually deployed. This is possible if some of the EcoWare components can be reused, and wired to multiple subscribers.

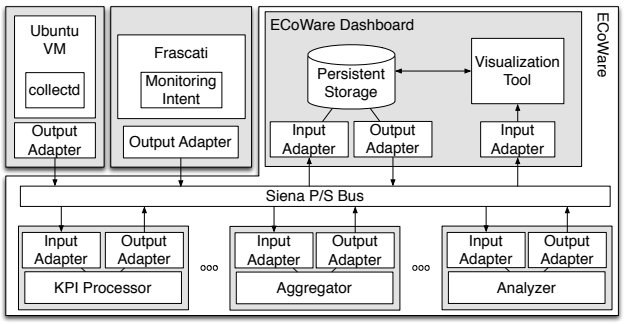

Fig. 2. The Event Correlation Middleware.

A. Probes

We currently insert probes into both the SCA execution environment and into the Ubuntu virtual machines on which the SCA composites are run. However, new probes can easily be added to support other service layers, such as the platform layer or the virtualization hypervisor.

[3]ECoWare is publicly available at http://home.dei.polimi.it/guinea/ECoWare/ECoWare.zip.

1) Probing SCA: We have developed a `Monitoring Intent` to insert probes into SCA composites. Intents are SCA's technology-agnostic way of defining component, service, and reference requirements, such as their need for specific security, reliable messaging, or transaction policies. In our case, the requirement is that the interactions undertaken by the component, service, or reference, need to be monitored. The Monitoring Intent is responsible for collecting and wrapping the collected SDOs into ECoWare events, and for sending them to the Siena P/S bus. It is also in charge of producing any low-level events that may be needed to calculate any required KPIs. To activate a Monitoring Intent one must reference the intent in the `requires` attribute of a component's SCA service or reference definition.

2) Probing Ubuntu Virtual Machines: The probes that we insert into our Ubuntu virtual machines are based on the collectd tool [11]. collectd is a system statistics collection daemon. It is implemented in C for performance and can easily be run on different architectures. It has an extensible library of plugins that make it suitable for collecting many different kinds of RIs, from CPU usage and memory usage, to network I/O, etc. As the data are collected they are sent to the Siena P/S bus for further management. Thanks to the Siena P/S bus we can easily replace the collectd tool with a different resource indicator probing tool, should the need arise (e.g., [12], [13]).

B. Data Processing

Data processing is provided by three different kinds of processors: `KPI Processors`, `Aggregators`, and `Analyzers`. These processors are used to implement $mlCCL$ KPI collection, data aggregation, and data analysis, respectively. RI collection does not figure here since once the RI SDOs are collected by their probes, they do not require further manipulation. The processors subscribe to events on the P/S bus, execute their internal logic, and publish their results back to the bus. This allows them to be strung together to obtain the information we desire.

`KPI Processors` are built using Esper [7], a tool for implementing complex event processing activities. Esper components execute queries defined using an Event Processing Language (EPL). EPL is an SQL-like language for developing event conditions, correlations, and aggregations. The difference between SQL and EPL is that, instead of running queries against stored data, Esper stores queries and runs data through them. The execution model is thus continuous rather than limited to the exact moment in which the query is submitted. We have defined appropriate EPL query templates for each of the KPIs supported by $mlCCL$. To include new KPIs in $mlCCL$ we need to create the required EPL query templates, so that they can be automatically configured during the transformation phase.

`Aggregators` are also built using Esper. They receive a variable number of event streams with the goal of creating a snapshot of aggregate data. The incoming streams consist of one `primary` stream, and at least one additional stream. For each of the additional streams the aggregator maintains a sliding window of incoming events to support the use of $mlCCL$'s `window` function. As soon as the primary stream sees a new event, that event is aggregated with the events being held in these windows. If one of the streams is empty, then the aggregated data is produced with a null value for that stream. For an aggregator with only one additional stream, that does not use the $mlCCL$ `window` function, the EPL is defined as:

```
SELECT *
FROM primaryName UNIDIRECTIONAL,
additionalName.win:length(1)
```

where `UNIDIRECTIONAL` indicates that the join is only performed when the aggregator receives an event on the primary stream, and the window for the additional stream is defined as a length window of 1 event. This way we only maintain and aggregate the last collected value for that stream.

`Analyzers` are responsible for implementing $mlCCL$'s data analysis features, including the data manipulation functions previously discussed. The analyzers are built using a simple assertion analyzer inspired by our previous work on WSCoL [1]. WSCoL is an assertion language for defining properties over XML snippets, and was developed in the context of our work on the monitoring of Web services. The analyzer requires that all the data needed to evaluate an expression be available in aggregate form on its incoming stream. This is why we always precede the analyzer with an aggregator configured to collect all the required SDOs.

C. The ECoWare Dashboard

The ECoWare dashboard is a Java Desktop application that supports both live charting, and on-line and off-line violation drill-down analysis. To enable live charting, the dashboard subscribes to the P/S bus to obtain streams of KPIs and RIs. Live charts are updated every time a new event arrives. The chart can also visualize multiple thresholds so that the manager can more easily identify problems.

The on-line and off-line violation drill-down analyses allow managers to visualize violations, together with any aggregations that were triggered by that violation. Off-line analysis requires that these data be made persistent, so that they can be retrieved on-demand. This is why, when we configure the dashboard with a new $mlCCL$ specification, its internal `Persistent Storage` component automatically subscribes to all the data that are relevant to that specification. We persist the data for 24 hours, and periodically cleanse the data that are no longer needed.

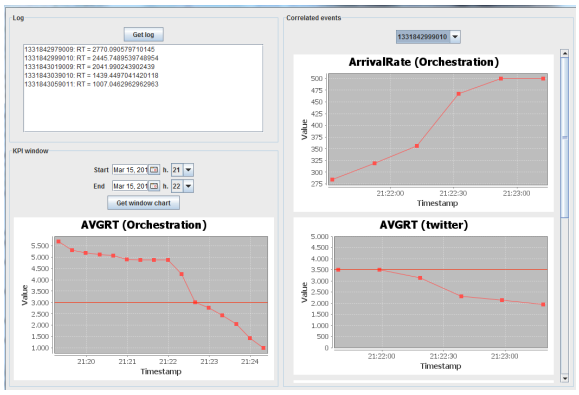

Fig. 3. The ECoWare Dashboard.

Figure 3 shows the dashboard's layout when showing a violation and its aggregated data. In the top left we have a log of violations, while in the bottom left we have a chart that is created to better show these violations. Finally, on the right hand side we have a series of charts that are generated when the user selects a single violation on the left. They show the SDOs that were aggregated to that violation.

IV. Evaluation

In this section we concentrate on two scenarios taken from our SocialTools example. In both scenarios the key dimension that we want to monitor is the average amount of time it takes the orchestrator to query Twitter, convert Twitter's output, and invoke the Weather service. In particular, we want a violation to be notified every time the average response time deviates too much from an expected range of values.

In the first scenario, every time there is a violation we want the system manager to be able to perform a drill-down analysis to see which of the following four services is causing the problem: the `Orchestrator`, the `DCA`, `Twitter`, or `WebserviceX.NET`. We also want the system manager to be able to look at the service's arrival rate.

```
O = (getUserWeather, TwitterWeather, Orchestrator);
DCA = (convert, Converter, DCA);
T = (getUserLocation, Twitter, Orchestrator);
W = (getWeatherByLocation, Weather, Orchestrator);

avgrtO = collect(avgRT, O, 20 seconds,
   2 minutes, null);
rateO = collect(rate, O, 20 seconds,
   1 hour, null);
avgrtDCA = collect(avgRT, DCA, 20 seconds,
   2 minutes, null);
avgrtT = collect(avgRT, T, 20 seconds,
   2 minutes, null);
avgrtW = collect(avgRT, W, 20 seconds,
   2 minutes, null);

violation = evaluate(avgrtO, avgrtO.get(value) > 5500
            || avgrtO.get(value) < 3000);

agg = aggregate(violation, rateO.window(2 minutes),
         avgrtDCA.window(2 minutes),
         avgrtT.window(2 minutes),
         avgrtW.window(2 minutes));
```

The average response time is calculated over short windows (`2 minutes`), and is output even more frequently (`20 seconds`), as to capture significant deviations in a timely fashion. When there is a violation ECoWare collects the `Orchestrator`'s arrival rate, and the average response times for the `DCA`, `Twitter`, and the `Weather service`, that were seen in the two minutes that led up to the violation. The average response times are setup with the same `pastWindow` and `outputRate` parameters to make them more easily comparable. The arrival rate for the Orchestrator considers the last hour of requests, and produces a new value every 20 seconds [4].

Figure 4(a)'s top chart shows that the Orchestrator's average response time is fluctuating. Horizontal dashed lines show the value's lower and upper thresholds. At a certain moment it starts dropping anomalously, to the point that it is clearly not executing correctly. In order to perform a drill-down analysis and discover the sources of this anomaly, we select one of the violations. For example, when we select the violation identified by an arrow in the figure, the Dashboard shows us its corresponding correlated data. These data appear in Figure 4(a) as the second and third charts. They are the Orchestrator's arrival rate and Twitter's Average Response Time. We clearly see that Twitter is also behaving irregularly. If we look at the Orchestrator's arrival rate we see that it had been growing very fast (up to 500 requests). The problem is that Twitter is blocking us out of its service. Twitter starts responding much more rapidly, with an error message, and the Orchestrator starts returning this error message directly to the caller, short-circuiting the rest of the composition. In this case the solution is to better throttle the requests that are made to Twitter.

Figure 4(b) shows a second scenario in which the Orchestrator's average response time starts growing anomalously. In this case, when we select one of the violations, we see that the problem is also visible in the average response time of the DCA, while Twitter and the Weather service are fine. Since the Orchestrator and the DCA are both hosted by SocialTools, the system manager decides to have a look at the infrastructure layer (see Figure 4(c)), using the following $mlCCL$ code:

```
DCA = (convert, Converter, DCA);
vm = (VM101);

avgrtDCA = collect(avgRT, DCA, 20 seconds,
              2 minutes, null);
memU = collect(vm, memUsed, 5 seconds, null, null);
cpuU = collect(vm, cpuUser, 5 seconds, null, null);
netIn = collect(vm, netIncoming, 5 seconds,
                null, null);
netOut = collect(vm, netOutgoing, 5 seconds,
                null, null);

violation = evaluate(avgrtDCA, avgrtDCA > 800);

agg = aggregate(violation, memU.window(2 minutes),
     cpuU.window(2 minutes),
     netIn.window(2 minutes),
     netOut.window(2 minutes));
```

In this case the pattern uses a violation in the DCA's average response time to trigger the collection of cpu, memory, and network performance attributes. If we choose one of the violations, we see spikes in the memory being used and in the user cpu of the underlying virtual machine. (In our example we artificially created the cpu and memory bottlenecks in the virtual machine running cpu- and memory-intensive background tasks.) This simulates the fact that multiple applications, not shown in this example, may be competing for the infrastructure resources.

A. Performance Evaluation

We have investigated the impact that data collection has on the running composite applications. We concentrated on the Monitoring Intent, since the overhead introduced by the collectd probe is, in comparison, negligible [11]. Siena, Esper, and our analyzers execute in parallel with the composite, and have no impact on its performance. Furthermore, they have demonstrated to be suitable solutions for systems that need to

[4]Such a long window is due to Twitter's public APIs, which allows up to 350 calls per hour before temporarily blacklisting the caller.

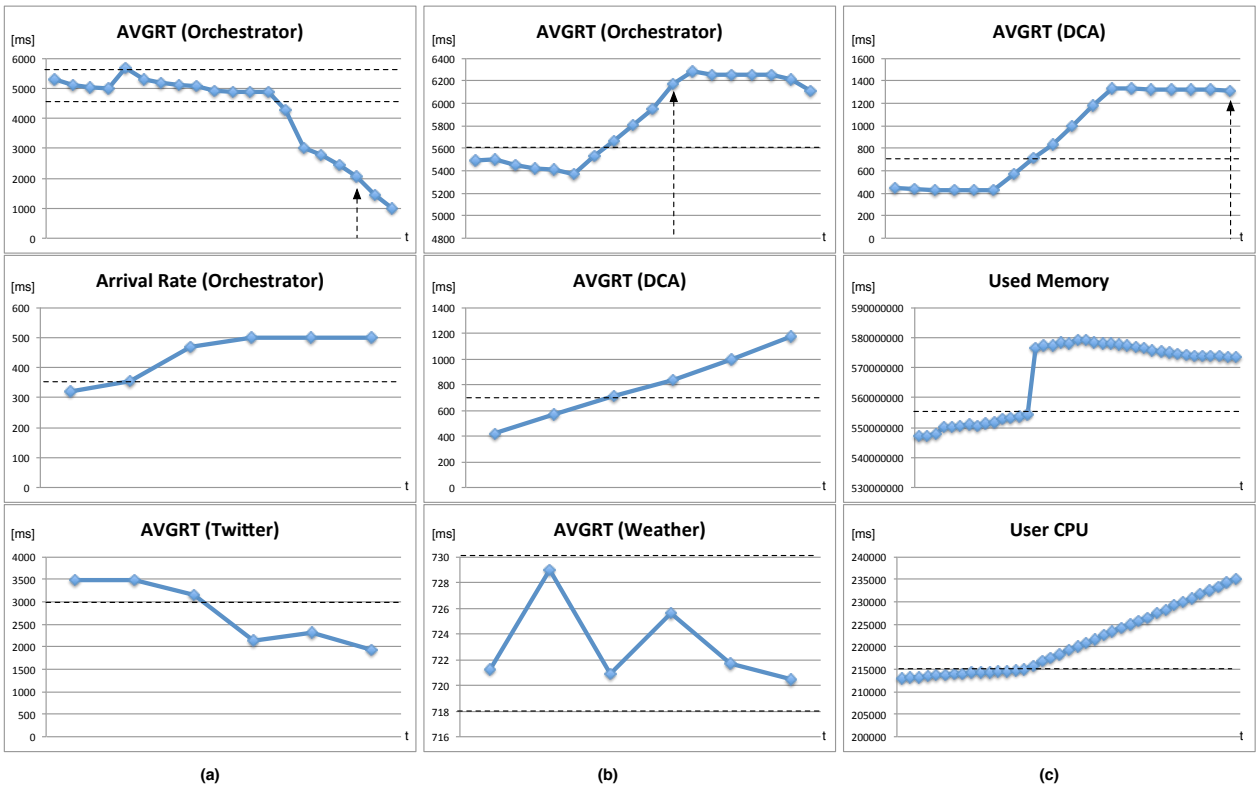

Fig. 4. Each column illustrates a drill-down analysis that is performed when a violation is chosen. In the top three charts we have a new average response time every twenty seconds (approximately six minutes of data are shown). The values on the y-axes are given in milliseconds. The violations that determine the three drill-down analyses are identified by vertical dashed arrows. Once a violation is selected, we are shown the KPI and RI values that were collected and aggregated with that violation. The x-axes of the bottom six charts correspond to two minutes, as defined by our $mlCCL$ specifications.

scale to high-volumes of events [14], [15], [1]. EcoWare was run on a local Windows 7 computer with a 2.20 GHz Pentium T4440 processor and 4 GB of RAM. The SCA composite was deployed to an Amazon EC2 Micro Instance, configured with a 64-bit Linux Ubuntu 10.04 OS, a CPU capacity of 2 EC2 Compute Units for short bursts, 613 MB of memory, and 8 GB of storage on an independent Elastic Block Storage volume.

TABLE I. MONITORING PERFORMANCE IMPACT

Setup	min	max	avg
0 intents	4498	5669	4897.26
1 intent	4551	5624	4870.3
2 intents	4521	5899	4870.93
3 intents	4546	5899	4870.98
4 intents	4525	6217	4957.73

Table I summarizes the impact that the monitoring intents had on our example. Five different tests progressively incremented the number of monitoring intents running simultaneously, from 0 to 4 (i.e. what is needed for our two scenarios). We ran each test 500 times. The runs were performed by 10 parallel threads, each with a random wait time of less than 1 second. The results show that all five average end-to-end invocation times are within a margin of one tenth of a second of each other. The performance hit due to the monitoring intents is negligible. Moreover, the activation of all four intents is not enough to make a noticeable difference when we take into account the amount of time it takes to serialize and deserialize SOAP messages in real-world scenarios.

V. RELATED WORK

IT Management solutions, such as IBM Tivoli [16], have long represented a key asset for software and system developers and managers. Cloud-based applications, and multi-layered systems in general, have however introduced new complexity. Some solutions, such as HyperSuite [17] and the CyMS Multi-Layer Management System [18], have been proposed. Yet they require high technical skills, and cannot easily be used by business experts. Furthermore, they create strong lock-in. In an attempt to simplify management we focus on the core notion of *service*: this allows us to provide a general-purpose monitoring solution that is easier to comprehend. We also strive to maintain a strong separation between $mlCCL$ and its runtime support, so that EcoWare can switch between different kinds of probing and processing tools with ease.

The Software and Service Engineering research communities have long focused on monitoring complex service-based systems. Although many approaches only concentrated either on the service layer ([19], [20], [2]) or on the infrastructure layer ([21]), multi-layer approaches have recently become the focus of various international research projects, due to the role they play in the broader context of multi-level governance.

The SLA@SOI EU project [22] proposed an open framework for managing service-based systems along a business layer, a software layer, and a virtualized infrastructure layer. Given an SLA they automatically constructed monitoring configurations for various monitoring engines deployed throughout the system [23]. However, they did not correlate vio-

lations at one layer with behaviors seen at others. The S-Cube European Network of Excellence on Software Services and Systems [24] established a cross-layer monitoring and adaptation methodology for service-based applications. They proposed a multi-layered version of the well-known Monitor-Analyze-Plan-Execute loop. The loop was then instantiated in a preliminary prototype that integrated different single-layer approaches in [5]. The integration focused heavily on BPEL processes, and suffered from a lack of well-devised abstractions that could guide the integration in a more general way. Emmerich et al. proposed Monere [6], an approach that takes into account the multiple layers that constitute modern Web applications. They instrument the system across all its layers, and exploit the BPEL workflows that compose the services to generate structural cross-domain dependency graphs. Then they monitor all the components and correlate their metrics to present the system manager with a comprehensive multi-layer diagnosis. Although similar to our work in scope, our focus is on a general-purpose multi-layer monitoring language, and on how to trigger data correlations at different layers, instead of on how to generate dependency graphs from BPEL specifications.

VI. Conclusion and Future Work

We presented $mlCCL$, an extensible language for defining how to collect, aggregate, and analyze runtime data in a multi-layered system, and extended ECoWare, an event correlation and aggregation framework. We also presented a visualization tool for live updates about a system's behavior, and for performing drill-down analyses of violations. Given that we have established the basic tools needed to perform multi-level service monitoring, in our future work we will focus on how $mlCCL$ and ECoWare could be used to perform more advanced on-line investigations (e.g., anomaly detection through event co-occurrence and clustering, root cause analysis, etc.). To do this, we will investigate the development of higher level $mlCCL$ macros or templates. We will also continue to evaluate our approach in the context of real-world systems, by directly engaging industrial partners. This will also require that we study the role that hypervisors have in managing cloud resources. Finally, we will study how to produce coordinated multi-layered recovery strategies that can allow us to keep complex systems running effectively and efficiently.

Acknowledgment

This research was funded by the EC, Programme IDEAS-ERC, Project SMScom (http://www.erc-smscom.org), and FP7 Project Indenica (http://www.indenica.eu/).

References

[1] L. Baresi and S. Guinea, "Self-Supervising BPEL Processes," *IEEE Trans. Software Eng.*, vol. 37, no. 2, pp. 247–263, 2011.

[2] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti, "Run-time Monitoring of Instances and Classes of Web service Compositions," in *Proceedings of the 2006 International Conference on Web Services*. IEEE, 2006, pp. 63–71.

[3] O. Moser, F. Rosenberg, and S. Dustdar, "VieDAME - Flexible and Robust BPEL Processes through Monitoring and Adaptation," in *Companion of the 30th international conference on Software engineering*, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, pp. 917–918. [Online]. Available: http://doi.acm.org/10.1145/1370175.1370186

[4] R. Popescu, A. Staikopoulos, P. Liu, A. Brogi, and S. Clarke, "Taxonomy-Driven Adaptation of Multi-layer Applications Using Templates," *Proceedings of the 2010 IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, vol. 0, pp. 213–222, 2010.

[5] S. Guinea, G. Kecskemeti, A. Marconi, and B. Wetzstein, "Multi-layered Monitoring and Adaptation," in *Proceedings of the 9th International Conference on Service-Oriented Computing, (ICSOC), Paphos, Cyprus*, 2011, pp. 359–373.

[6] B. Wassermann and W. Emmerich, "Monere: Monitoring of Service Compositions for Failure Diagnosis," in *Proceedings of the 9th International Conference on Service-Oriented Computing, (ICSOC), Paphos, Cyprus*, 2011, pp. 344–358.

[7] EsperTech, "Complex Event Processing," http://esper.codehaus.org, 2010.

[8] G. Cugola and A. Margara, "Tesla: a Formally Defined Event Specification Language," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM, 2010, pp. 50–61.

[9] L. Baresi, M. Caporuscio, C. Ghezzi, and S. Guinea, "Model-Driven Management of Services," in *Proceedings of the Eighth European Conference on Web Services, ECOWS*. IEEE Computer Society, 2010, pp. 147–154.

[10] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, Aug. 2001.

[11] collectd, "collectd – The System Statistics Collection Daemon," http://collectd.org/, 2012.

[12] S. Bhatia, A. Kumar, M. E. Fiuczynski, and L. Peterson, "Lightweight, High-resolution Monitoring for Troubleshooting Production Systems," in *Proceedings of the 8th USENIX Conference on Operating systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 103–116. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855749

[13] I. Agarwala and K. S. O. Computing, "Sysprof: Online Distributed Behavior Diagnosis through Fine-grain System Monitoring," in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006), p. 8, IEEE Computer Society*, 2006.

[14] A. Carzaniga and A. L. Wolf, "Forwarding in a Content-Based Network," in *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, Aug. 2003, pp. 163–174.

[15] EsperTech, "Esper performance," http://docs.codehaus.org/display/ESPER/Esper+performance, 2007.

[16] IBM, "Tivoli Software solutions: IBM Integrated Service Management for a Dynamic Infrastructure," http://www-01.ibm.com/software/tivoli/solutions/, 2012.

[17] eBusiness 1, "HyperSUITE Monitoring Framework," http://eb1.biz/hypersuite, 2012.

[18] Cyan, "CyMS Multi-Layer Management System," http://cyaninc.com/cyms/cyms-multi-layer-management-system, 2012.

[19] G. Kaiser, J. Parekh, and P. Gross, "Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems," in *Proceedings of the 5th Annual International Workshop on Active Middleware Services*, 2003, pp. 22–30.

[20] G. Spanoudakis and K. Mahbub, "Non-intrusive Monitoring of Service-based Systems," *International Journal of Cooperative Information Systems*, vol. 15, no. 3, pp. 325–358, 2006.

[21] Resources and Services Virtualization without Barriers, http://www.reservoir-fp7.eu/, 2011.

[22] SLA@SOI, "Empowering the Service Industry with SLA-aware Infrastructures," http://sla-at-soi.eu/, 2011.

[23] H. Foster and G. Spanoudakis, "Advanced Service Monitoring Configurations with SLA Decomposition and Selection," in *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan*, 2011, pp. 1582–1589.

[24] S–Cube Network of Excellence, Software Services and Systems, http://www.s-cube-network.eu/, 2012.