Run-Time Correlation Engine for System Monitoring and Testing

Viliam Holub
Lero, System Research Group
University College Dublin,
Ireland
viliam.holub@ucd.ie

Trevor Parsons
Performance Engineering Lab.
University College Dublin,
Ireland
trevor.parsons@ucd.ie

Patrick O'Sullivan
WPLC SVT
IBM Software Group
Dublin, Ireland
patosullivan@ie.ibm.com

John Murphy
Performance Engineering Lab.
University College Dublin,
Ireland
j.murphy@ucd.ie

ABSTRACT

Today's enterprise applications can produce vast amounts of information both during system testing and in production. Correlation of this information can be difficult as it is generally stored in a range of different event logs, the format of which can be application or vendor specific. Furthermore these large logs can be physically distributed across a number of different locations. As a result it can be difficult to form a coherent understanding of the overall system behaviour. This has implications for a number of domains (e.g. autonomic computing, system testers), where an understanding of the system behaviour at run-time is required (e.g. for problem determination, autonomic management etc.)

This paper presents an approach and implementation of run-time correlation of large volumes of log data and symptom matching of known issues in the context of large enterprise applications. Our solution provides for automatic data collection, data normalisation into a common format, run-time correlation and analysis of the data to give a coherent view of system behaviour at run-time and a symptom matching mechanism that can identify known errors in the correlated data on the fly.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Diagnostics; C.4 [**Performance of Systems**]: Reliability, availability, and serviceability

General Terms

Performance, Reliability, Verification

Keywords

System Monitoring and Testing, Event Correlation, Event Filtering

ICAC-INDST'09, June 16, 2009, Barcelona, Spain.

1. INTRODUCTION

Current enterprise systems are often very complex, consisting of large numbers of software components that can be physically distributed across different hardware devices (Figure 1). Each software component (e.g. web server, application server, database etc.) will in general produce numerous logs files containing information on the system level events. The volume of information produced for typical enterprise applications can be extremely large. This is especially the case where systems are expected to cater for high numbers of simultaneous users/requests. Furthermore, while logging facilities are common practice today, most are application or vendor specific. Events from different components can differ in their format and level of detail. This can be particularly problematic for manual or automatic correlation and analysis of the data. Very often the correlation of such data is required when issues arise during testing to obtain an understanding for global system behaviour. Aggregating several sources of information represented in various formats can be very time consuming and often requires the attention of highly skilled engineers. Due to the time required for correlation and analysis, this activity is generally performed manually, after test runs have completed. As a result system testers often do not have a global view of these events as the system executes. This can be particularly problematic for timely root cause analysis which can require analysis of an issue within its running context.

This paper presents an approach and implementation for run-time correlation and symptom matching in the context of large enterprise applications. Our solution provides for (a) automatic data collection, (b) data normalisation into a common format, (c) run-time correlation and analysis of the data to give a coherent view of system behaviour at run-time and (d) a symptom matching mechanism that can identify known errors in the correlated data on the fly. We address a number of performance issues that can arise when gathering, normalising, correlating and analysing large volumes of information. We have implemented our approach in the form of the *Run-Time Correlation Engine and Symptom Database* (RTCE) and have applied it in a real industry test environment. Our performance results show how our approach is suitable for the correlation of data from real enterprise systems.

Our solution has been developed as a result of a real industry requirement in the context of system testing of large enterprise applications. However we believe it is applicable in a wide range of domains (e.g. for live system administration, monitoring and problem determination). Furthermore we believe it is particularly

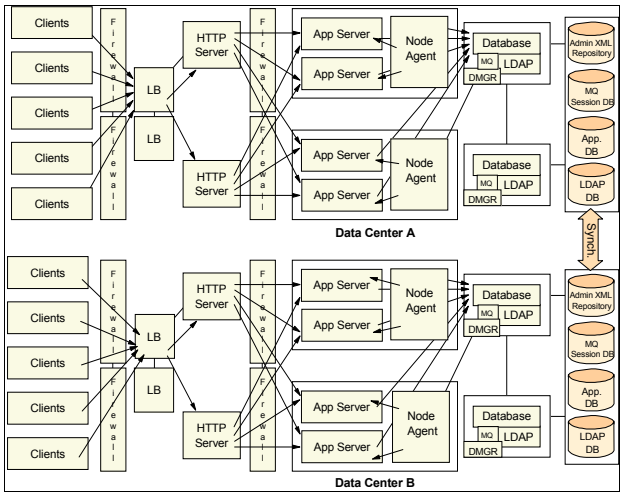


Figure 1: Example of a distributed enterprise system (from [3])

relevant in the context of autonomic computing [1] where large volumes of information may need to be correlated and analysed on the fly. Autonomic management systems, for example, require an understanding of global and local system behaviour such that they can self-manage and adapt to system events in a timely manner [2].

The remainder of this paper is structured as follows: Section 2 outlines a number of criteria that we feel must be met for our approach to be considered useful. These criteria have been gathered from discussions with a number industry testing teams. Our work builds on top of several technologies applicable to the RTCE project which form part of IBM Autonomic Computing Toolkit [4][5] (AC Toolkit). We present these technologies in detail in Section 3. Section 4.1 presents our overall system architecture and gives details on the core components of the Run-Time Correlation Engine and Symptom Database. Section 5 gives our performance results. Here we present performance analysis of the main subcomponents involved in the RTCE architecture and we show how these components scale to handle increasing numbers of events. In Section 6 we present related work and Section 7 gives our conclusions.

2. EVALUATION CRITERIA

In order to fully understand the requirements for run-time correlation and analysis of event logs of enterprise applications we have held discussions with numerous industry testing teams that are responsible for performance and reliability tests carried out on real enterprise systems on a daily basis. From these discussions we have produced a number of evaluation criteria for assessing our work. The evaluation criteria are the attributes which we (and industry test teams) believe are required, in order for our approach to be considered useful. In this section we outline these evaluation criteria and briefly discuss how they have been addressed by this work.

When dealing with large volumes of data that is to be aggregated, correlated and analysed on the fly performance is an important factor. The following criteria were identified as critical for the validation of the RTCE

Responsiveness: It is critical that the responsiveness of the RTCE does not exceed acceptable limits. In particular the responsiveness in terms of the run-time correlation, run-time log searching, and run-time symptom identification is of utmost importance for users/clients of the RTCE.

Overhead on monitored environment: It is crucial that the performance impact on the host system is minimised. Low overhead is required in particular for production monitoring. However, this can be a requirement even in test environments where monitoring overhead can introduce issues for system and performance testers. For example, distorted test results, such as reduced response time, lower system capacity etc.

Scalability: Typical enterprise application can contain large numbers of components and can be expected to handle high user loads. As such scalability of the RTCE (in in terms of the number of components that the system can handle and the amount of information contained in the variety of logs) is important. A high end user capacity is also a requirement for large system test teams that require simultaneous access to the correlated data.

Our performance evaluation section (see Section 5) analyses the RTCE in terms of scalability, responsiveness and performance overhead. Along with these attributes, we have identified several desirable features that resulted from discussions with potential end users (i.e. system and performance testers).

- *Filtering* is very much desirable such that the volume of data to be analysed can be reduced. The RTCE allows for events to be filtered per component as well as the centralized, correlated log. There are numerous filtering properties, for example component ID, severity, pattern-match, etc.

- Reliability test runs of enterprise applications can typically last for a number of days resulting in a large amount of high volume log data. Upon completion understaning whether a system has passed or failed a reliability run can be a challenging and time consuming task. Understanding of system level log information can help testers to understand if errors occured during a run. The RTCE provides a *log healh report* through statistical analysis of log data giving informaiton on the number and distibution of events that have occured occurred. Performance counters such as events produced per component also allow for an understaning of the rate of information being produced per component.

- *Offline log analysis* is provided to allow for post test analysis. The feature allows for instant access to correlated data at any point during test run. This allows for user/clients to understand the state of the different system components at different points during system exeuction.

- Clocks on local testing networks are usually well synchronized, but that does not hold for wide networks. The system should be aware of this and as such there is a requirment for a *clock synchronization mechanism*.

The above features are described in more detail in Section 4.

3. BACKGROUND

The *Run-Time Correlation Engine and Symptom Database* implementation (described in detail in Section 4.1) builds upon a number of current technologies. This section describes existing autonomic technologies applicable to the RTCE project which have already been built as a part of IBM Autonomic Computing Toolkit [4][5] (AC Toolkit). More precisely in the following subsections we introduce the Common Base Event logging format, the Generic Log Adapter technology, the Agent Controller for data collection and the Symptom Database for the automatic identification of known issues.

Common Base Events [6] (CBE) is a common XML format that can be used for the representation of system events. The CBE format contains information on the *source component*, reporting component and data associated with the event (e.g. time, event message, sequence number). The source component is the component where the event occurred, whereas the *reporting component* is considered the component which observes the situation. For specific attributes which are not reflected in Common Base Event format, an `ExtendedDataElement` entry can be used and forms part of the CBE format. Proprietary events messages can be converted to CBE using Generic Log Adapters described in the following subsection. In fact, since OASIS adopted CBE format as a base for the WSDM Event Format [7] (WEF), CBE can be seen as an implementation of the WEF standard.

The Generic Log Adapter (GLA) [8] overcomes problems with proprietary event formats by automatically converting them to the CBE format. The IBM Autonomic Computing Toolkit [4][5] provides GLA configuration adapters to support over 280 log types. To achieve high flexibility of accepted event format, the GLA is organized in several chained layers responsible for obtaining events, understanding, converting, and passing for further processing [8]. Furthermore, the GLA technology is highly configurable through an XML configuration file called an *adapter*. An adaptor specifies properties of each layer including the class name of the actual GLA implementation. The GLA processes the adapter during initialization and instantiates all layers. Moreover, an adaptor allows for the definition of more than one context for a particular GLA, thus more than one log file format can be processed by a given GLA. The Adapter Rule Builder presented in the literature [4] can be used to construct adaptors.

The Agent Controller is a daemon process which allows for control of remote agents which reside across the host system. The agent controller can remotely start and stop the remote agents and manages communication between the agents. The agent controller was originally developed as a data collection engine, and is part of the Eclipse TPTP project [9].

Agents are responsible for gathering specific information from the host system. Each *agent* is represented by a binary file which upon execution provides services for *clients*. Clients are typically user's application components (e.g. database, web server, application server etc.) which produce information in the form of events for agents. Each agent connects to one or more Agent Controllers either locally or remotely and forwards the required information.

The symptom database [10] provides a mechanism for matching known problems in large volumes of data. It allows for the documentation of expert knowledge and for the analysis of logged events using this expert knowledge. Known issues can be represented in the Symptom Database as a set of symptoms in XML format. Each symptom contains three pieces of information: situation, solution, and resolution (see the example in Figure 2). The *Situation* describes the context of the symptom. The situation is described by a set of patterns that can be matched in the event logs. The *Solution* gives the reason or reasons why an error has occurred, while the *Resolution* gives details on as to how to rectify the error. For system testers this may be a manual fix, while for autonomic management systems the resolution may be performed automatically. System databases can be created manually by system experts or automatically using a symptom database builder [11].

4. RTCE DESIGN

In this section we present our approach for run-time correlation of log data and symptom matching. Our approach has been realised as part of the *Run-Time Correlation Engine and Symptom Database* implementation. In the following subsections we detail the implementation architecture. More precisely in Subsection 4.1 we describe the general approach for run-time correlation using the architectural diagrams and explain how the components that make up the overall architecture interact. Subsection 4.2 explains in more fine grained detail the internals of the RTCE and how we achieve run-time correlation and symptom matching. Finally in Subsection 4.3 we detail how the correlated information and analysis of the data is presented to the tool user.

Situation:

```
XMLC0005E
admin
java.rmi.RemoteException:
StaleConnectionException:
SQL1224N
55032
```

Solution:

```
This error has been observed with installations
using DB2 on AIX when DB2 is configured to access
the WebSphere repository directly using shared
memory (default) instead of TCP/IP loopback. This
severely limits the number of available
connections.
```

Resolution:

```
To see if this is the case, run the db2-mode
command "LIST DATABASE DIRECTORY" and view the
entry for the WebSphere repository database.  It
should show the database as an alias for a
database with a different name, and an access
type of "remote".  If this is not the case,
contact your database administrator and have your
websphere repository accessed as a remote database
via TCP/IP.
```

Figure 2: Simplified example of a symptom (from the symptom database for the WebSphere Application Server Advanced Edition.)

4.1 RTCE architecture

A schematic diagram of the RTCE architecture is depicted in Figure 3. This diagram shows the interactions between the main RTCE components i.e. the host components, the host log files, the Generic Log Adapter, Agent Controller, the RTCE core and presentation components. Next we will explain how the different components that make up the architecture interact when deployed by walking through the scenario depicted in Figure 4.

Figure 4 gives a application consisting of two Apache servers and one MySQL database server. Both Apache processes and the MySQL processes related to the components in Figure 3. Components store events in logs using their proprietary formats. In general, a component can store events to an arbitrary number of logs. For example, the Apache usually logs successful requests to `access.log` file and erroneous requests to `error.log` file, while MySQL stores erroneous requests to the `mysql.err` file.

In this example the underlying operating system is also considered a component. Events reported by the operating system can be very valuable for problem determination [1][4]. In our example, events from the operating system are stored in the `messages` file. Events from the various logs are converted to the CBE format by Generic Log Adapters (GLA). A single instance of a GLA can convert events from multiple log files.

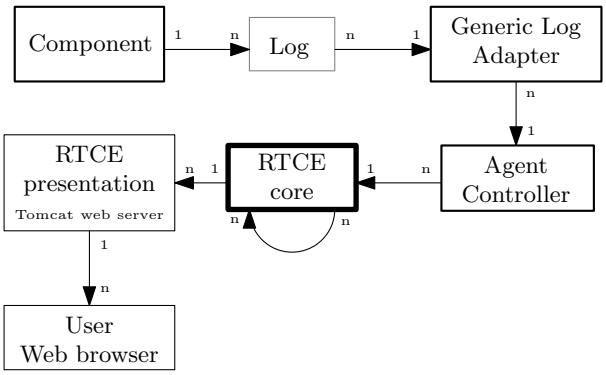


Figure 3: RTCE architecture

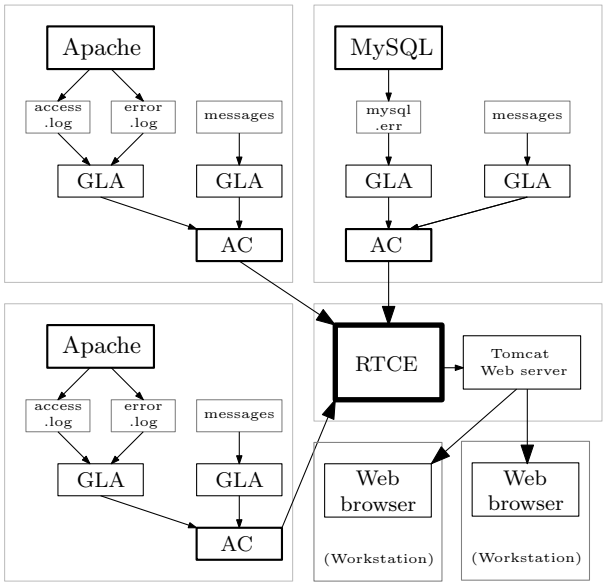


Figure 4: Application and RTCE deployment example

A single Agent Controller (AC) is deployed per host application. The Agent Controller is an intermediate between local GLAs and the RTCE core; it routes events from specified GLAs to the RTCE core. The RTCE core maintains an open connection with all available Agent Controllers, collects incoming events, and processes them. Behavior of the RTCE core depends heavily on running *contexts* (Section 4.2). A context defines which components are to be monitored, specifies the level of filtering, and other host application specific properties. Contexts are useful in larger environments, where more than a single application may be deployed within a given environment (i.e. across a given network shared by a large number of applications). Together with the RTCE core, we deploy the RTCE presentation component on the Tomcat Web server. The role of RTCE presentation component is to accept user requests, obtain required information from the RTCE core, and present this information in a convenient way through dynamic web pages (see Section 4.3 for details). To allow for shared access to the data users can interact with the RTCE through a web browser (see Section 4.3). This is particularly useful for system testers that want to gain an understanding of the current state of the system.

4.2 RTCE core

The internal structure of the RTCE core is displayed in Figure 5. In this subsection we explain how the various subcomponents within the RTCE core interact. The whole internal scheme of the RTCE core is constructed with well-defined interfaces. Components are implemented as pluggable classes which makes the RTCE core flexible for modifications and extensions.

4.2.1 Event aggregation

The *Agent Controller Manager* (ACM) is responsible for managing connections with Agent Controllers. Agent Controllers are auto-discovered in the network or are explicitly specified in the RTCE configuration. The auto-discovery feature scans local network for available Agent Controllers (AC) by systematically pooling specified IP address space. The main advantage of the auto-discovery feature is that it allows the RTCE core to set itself up in the network automatically with minimal intervention of the network administrator (self-configuration).

For each Agent Controller in the network, the ACM instantiates an *AC Connection* (ACC). ACC maintains the connection between the RTCE core and the Agent Controller. In a case of connection loss, ACC tries to re-open the connection. ACC periodically retrieves a list of available adapters from the Agent Controller. If a new GLA appears, ACC consults the Configuration to assess whether data from the new adapter is of interests (i.e. defined in the relevant context, described later in this section). If so, ACC initializes a *Data Processor* which starts to collect events from the adapter. A Data Processor decodes events from remote GLA and passes them to the *Filter*.

The *Filter* component applies specified filters to incoming events in order to discard events that are deemed to be redundant or irrelevant. We have implemented a powerful filtering scheme called a *composite filter*. A *basic filter* is based on matching of selected CBE properties with a regular expression or a numeric value. A composite filter is a set of basic and composite filters organized in a logical tree. Each filter, when matched, marks the particular event as *deny* or *allow*. Filters are applied in depth-first search (DFS) order where a descendant filter is considered to be a refinement of the ascendant filter. This ordering allows for a fine-grained selection of events. Events marked as deny are removed.

Subsequently, the filter passes the event to the *Symptom Identification*.

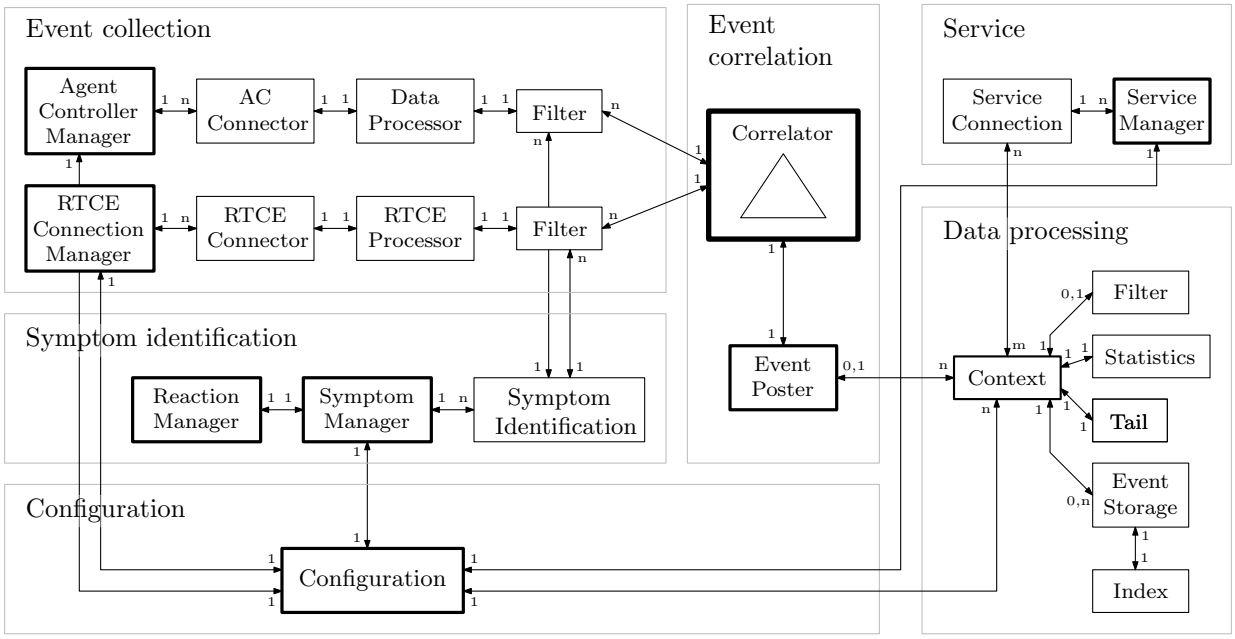


Figure 5: RTCE internal structure

4.2.2 Symptom matching

For every symptom database that is applied to the event information, the *Symptom Manager* maintains a *Symptom Identification* component. The Symptom Identification is applied for matching incoming events with known issues. When the event is matched with a particular symptom situation, it is augmented with a corresponding explanation and a solution. Events matched against specified critical symptoms are passed to the *Reaction Manager* for further immediate reaction. We do not implement any reaction functionality itself, but rather provide an open interface for additional external plug-ins, such as an automatic recovery, sending a notify to system administrators, or creating a ticket in issuse tracking system.

Considering the high amount of symptoms in a database (for example, the symptom database for the WebSphere Application Server 6.1 currently contains 18828 symptoms), we have implemented a specialized pattern-matching algorithm for fast symptom identification based on Aho-Corasick [12] search (more in Section 5).

We are aware that the symptom matching is limited to only one specified component in the system. This limitation is inherent to current symptom databases. In the near future, we plan to define a general interface for the symptom matching which would allow precise and statistical analysis of relations among different components.

4.2.3 Correlation

Due to the possibility of delays when aggregating events over the network, the *Correlator* component sorts incoming events according to their timestamp and sequence number (part of CBE). Right order of events is particularly important later for causality and trace analysis. To stabilize the event's position within the correlated log events must be kept in the Correlator for a specified amount of time (a *correlation window* (otherwise events which were delayed may require the logs to be re-correlated). Typically, event position is stabilized within few seconds for most of the applications.

Although hardware clocks are usually perfectly synchronized in co-located computer clusters, current enterprise systems may be deployed in part in geographically different locations. For situations where the synchronisation of the host application's hardware clocks is not optimal, a time delay resolution mechanism is being developed. The time delay resolution mechanism will reside with the centralised RTCE core and will resolve the delay between the different clocks by polling each of the hardware devices periodically.

4.2.4 Data processing

After correlation has occurred the *Event Poster* is responsible of removing the events from the Correlator. The Event Poster passes events to the data processing section. The Data processing section provides a number of mechanisms for summarising the correlated data and for processing the data such that it is consumable by clients of the RTCE (e.g. system maintainers/testers). The data processing component receives data from the Event poster and passes the information to *Context* components. A context defines what information (and how) should be presented to the tool user from the correlated log. Most often, a context defines the components that make up entire host application or a subsystem of the host application. Essentially, a context defines which components of the system are interesting for the user to analyse. One or more contexts can be defined for a given host application.

A second point of filtering is available for user configurable filtering of the correlated data. To give a high level view of the log a *statistics* gathering mechanism has been developed. This allows for a summary view of the correlated log. The various default statistics include histogram of event severities, first and last event occurrence, total number of events, average number of events per second, and the distribution of events. The statistical summary is particularly useful for system maintainers as it can give a very good indication of the host system health.

A *tailing* mechanism is also provided for by the data processing component. This mechanism allows for tailing of the correlated data and is particularly important for system tester who wants to

monitor correlated system events in real time. Tail is a FIFO buffer of events kept in memory.

The *Event Storage* mechanism stores events permanently in a file for historic and off-line system behaviour analyses. Since the size of output data may be enormous, locating events by searching the whole file would be ineffective. To overcome this problem we have built a light-weight *Index* based on event occurrence time and position in the output file. Requests for events from the particular time frame are then retrieved instantly from the exact position.

4.2.5 Service and Configuration

The *Service Manager* handles incoming connections from servlets in the RTCE presentation component or from other RTCE instances. For every open connection, the Service Manager instantiates a *Service Connection*. The Service Connection accepts incoming requests, retrieves required information from the system, and sends the response.

Finally, the *Configuration* is a container of user-specified and system-specified options. User-specified options are uploaded from the configuration file at start-up and updated in real time through the Service Connection. Components can also store configuration options for other components. For example, a Context can save a list of required application components, which is later used by the AC Connector when deciding which adapter should be monitored.

4.3 User interface

For the RTCE we have used AJAX-based web pages to provide rich user interface. The user interface is implemented in two parts. The first part resides on the Tomcat Web Server in a form of servlets. Servlets accept AJAX requests from clients (web browsers), obtains necessary information from the RTCE core, and respond to the requests by presenting the correlated log information in a number of meaningful formats (e.g. log tail, full-correlated log off-line access, statistics, matched symptoms etc.).

The second part runs in user's web browser in a form of a JavaScript dynamic page. JavaScript manipulates DOM elements of the web page in order to quickly and in real time automatically augment the users view with newly updated information.

5. PERFORMANCE EVALUATION

In this section we discuss how we have validated our approach in respect of the evaluation criteria outlined in Section 2. Our approach was to understand the performance characteristics of various RTCE components in terms of response time and resource consumption under varying loads. This allowed us to understand the various component response times, scalability and resource consumption. Furthermore from our tests we discovered potential bottlenecks to scalability. The bottlenecks were addressed through component optimizations and the RTCE system performance was reevaluated.

5.1 Test environment

All our tests are run on a relatively modest reference machine Intel Xeon 5140 2.33GHz, 8GB RAM, RedHat Enterprise Linux SE release 4, and the Java virtual machine IBM J9 VM 2.4.

For our test purposes a single GLA adapts events from the Apache access log. We measure time requirements of all the RTCE components involved in the system. To obtain a representative picture, we have initiated all the components to a typical deployed configuration state. The initial filtering consists of four composite filters. The symptom database has been initialized with 50000 symptoms. The Event poster keeps a 2 second correlation window (at a load of 1000 messages per seconds that corresponds to about 2000 events in the correlation heap). The output filter consists of four pattern-matching rules. Events are stored in the output file on a local hard drive together with the index.

RTCE component	CPU usage in μs
GLA	1151.38
Correlator	149.89
Event Storage	127.10
Symptom Identification	26.93
Data Processor	20.92
Context Filter	20.50
AC	12.30
Component Filter	5.12

Table 1: Average CPU usage per event and RTCE component

5.2 CPU consumption

Figure 6(a) and Table 1 shows average[1] amount of time spent by one event in RTCE components passing through the system from the source to the output file (components with minimal impact have been removed).

Evidently the most demanding component in the architecture is GLA which consumes almost tree times more CPU time than all the other components (memory usage of a GLA for the Apache access log is about 54MB). We address GLA performance in Section 5.3.

Correlator is the only single point of synchronization in RTCE and is a performance critical element since increasing the level of parallelism does not scale up. To maximize scalability, the Correlator uses a heap data structure organized according to the event timestamp. As displayed in Figure 6(c) (Table 2) the Correlator scales very well and does not constitute a performance problem.

The Event Storage spends most of the time in object serialization. The process of saving events on a relatively slow hard drive have only marginal impact on performance since all I/O operations are performed in a separate thread, isolated from the real-time part. Theoretical barrier for scalability of the Event Storage is the speed of writing operations. Contemporary hard drives are able to write data at a throughput of more than 100MB/s, which sets the limit to about 80000 events per second for a single hard drive machine.

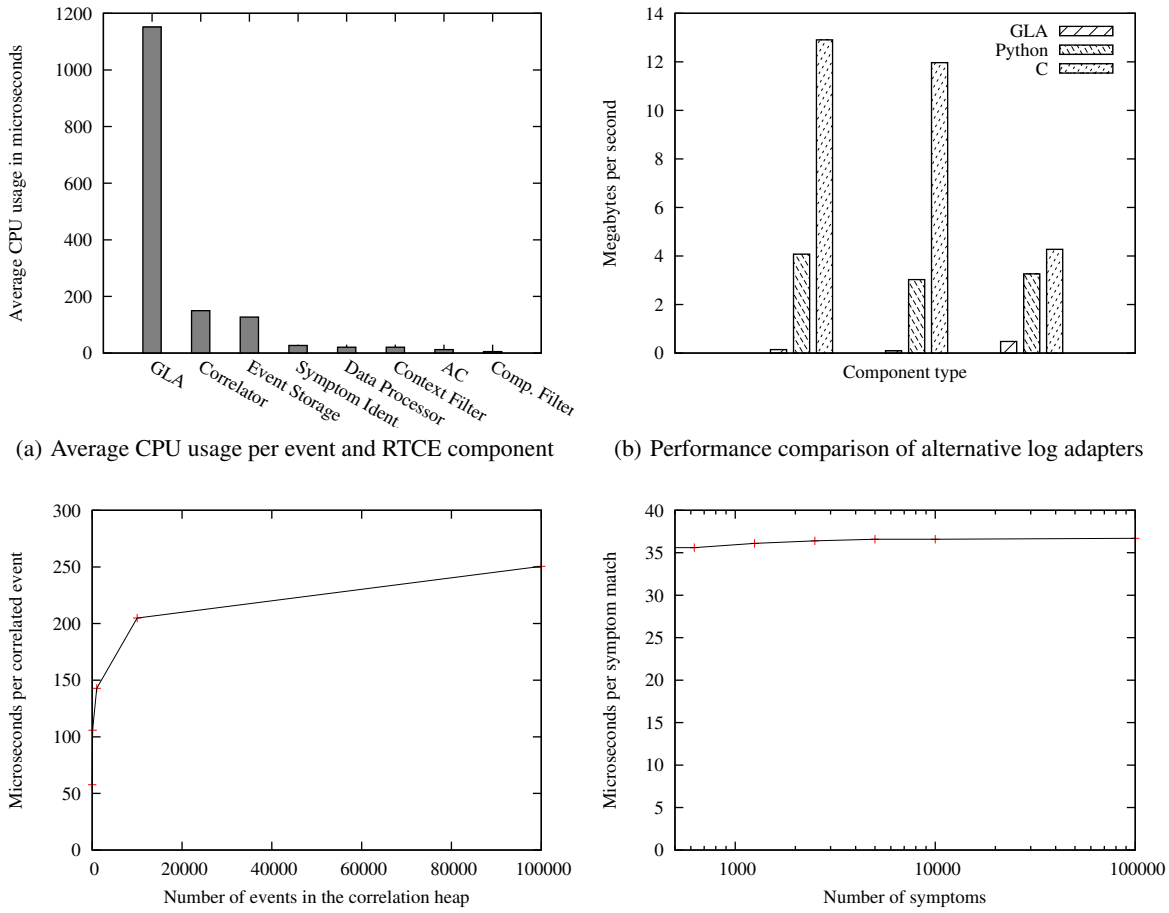
Symptom databases may grow substantially, more that 10000 records is not an exception. Therefore, the Symptom Identification have to recognize symptoms in real time even though there may be tens of thousands patterns to match. After having problems with performance of the Java regular expressions, we have implemented a specialized solution based on the string searching algorithm Aho-Corasick [12]. Measured results displayed in Figure 6(d) (Table 3) show excellent scalability in a number of symptoms; the time complexity of the symptom identification is almost independent of the database size.

5.3 GLA performance

The GLAs make heavy use of regular expressions to allow for easily configurable generic parsing of unstructured data. Unfortunately, because of the inherent length and complexity of the rules, the parsing stage has a major impact on the overall performance. This problem however can be addressed in several ways, either by creating static parsing code [13], careful optimization of regular expressions [14], or more esoteric approaches such as gridifying [15].

High CPU utilization of GLA is undesirable since the load introduced could negatively impact responsiveness of the components

[1]Average of 10^5 samples.



(a) Average CPU usage per event and RTCE component

(b) Performance comparison of alternative log adapters

(c) Scalability of the correlation process

(d) Scalability of the symptom identification

Figure 6: Scalability and performance plots

Heap size	10^1	10^2	10^3	10^4	10^5
Correlation of one event in μs	57.7	105.9	142.8	204.9	250.5

Table 2: Scalability and performance of the correlation component

Symptom database size	312	625	1250	2500	5000	10000	100000
Symptom match in μs	35.6	35.6	36.1	36.4	36.6	36.6	36.7

Table 3: Scalability of the symptom identification

	GLA/Java		Python		C	
	kB/s	msgs/s	kB/s	msgs/s	kB/s	msgs/s
Apache access	144	1242	4077	35072	12908	111048
Apache error	94.2	802	3027	25774	11966	101898
WebSphere AS	476	1590	3265	21156	4274	27698

Table 4: Log adaptation alternatives

in production. In situations where the overhead of running GLA on the same host as the component are not acceptable, the host application logs can be stored on network attached storage (NAS) devices. Consequently, a GLA can be deployed on a different host, eliminating potential side-effects to the monitored application.

As an alternative approach, we have implemented new adapters specialised for a particular log format in Python and C. Benchmarks of alternative adapters together with original GLA are presented in Table 4 and Figure 6(b). The implementation in Python is usually more than 7x faster for simple logs and up to 50x faster for more complicated logs. The significant speed-up makes the Python a viable alternative. From our experience, the time required for the implementation of a Python adapter using regular expressions is comparable or smaller to the construction of a GLA adapter. Adapters in C have superior performance characteristics, being up to 90x faster than GLA, but from our experience the implementation in C is significantly lengthly and error-prone.

5.4 Stress tests

Finally we have also performed stress tests on the reference machine which reveal that the maximum continuous load RTCE is able to handle in a typical configuration is 5000 events per second. When overloaded, RTCE is not able to correlate events within the correlation window, resulting in errors in event ordering. From discussions with experts from the domain of system testing and diagnosis, we know that the typical hosted enterprise application operated in a network cluster produces about 400 events per second in peaks. More events are rather unusual since the quality of reported information quickly decreases with quantity. Therefore, taking also into consideration user's actions, a safe bet for the RTCE is about 4000 events per second on average. Given 400 events per second per application, the RTCE is able to safely handle more than 10 applications simultaneously on a dedicated hardware.

6. RELATED WORK AND DISCUSSION

In this section, we summarize related projects from various backgrounds and discuss their relations to the RTCE. While there is a lot of prior work in event collection and analysis, we believe that RTCE is exclusive in the combination of various technologies into one coherent environment.

Event correlation [16, 17] is an analysis technique for mass event processing which role is to extract the most important information from the stream of low-level events and present them in more meaningful way. It has been used mainly in (on in relation with) network and system management, not so in heterogeneous software management though. From our perspective the main difference in event correlation between network and software management is in the background information the event carry. The network management usually produces a vast amount of measuring event (probes) which information value rapidly declines over time. In contrast, software systems produce events which describe behavioural steps. Therefore in a case of system malfunction historic retrospection is valuable for root cause analysis even though the preceding historic events may look innocent. That is not typically the case in network management where the error is detected immediately by alarms.

In Unix systems, the light-weight daemon *syslogd* is responsible for collecting and basic filtering of system events. Events, passed through local socket, consist of a priority (severity in CBE), a facility (a numeric identifier similar to component ID in CBE), and a message itself limited to 900B. Events are matched against simple rules based on priority and facility and saved in files specified. Alternatively, syslogd can pass all the events to the specified host over a (unreliable) UDP connection, without any filtering though. Because of a relatively weak support for network logging and filtering of the original syslogd, many improvements have been introduced to the original concept in various projects. For example, *syslog-ng* [18] adds filtering using regular expressions, reliable TCP connection, and secure communication over SSL/TLS.

Thanks to its simplicity, the class of syslogd-like approaches offer a relative high performance. However, the simplicity is traded off for many features required in contemporary systems. Probably the main practical obstacle is the limited support for user (non-system) applications. Components can be identified by 12 predefined and only 8 user available numeric identifiers, making almost impossible to distinguish component instances even for mediocre large systems. Moreover the lack of better message categories forces application developers to use their own, application-specific, vendor-specific, and language-specific logging approach, thus introducing many incompatible formats.

Scribe [19] is an open-source project for collecting system events designed for high scalability in wide networks and resilient to network failures. Scribe server runs on every node, accepts incoming events and resend them in buckets to the centralised server. The centralised server stores events in files specified or resent them to another layer of servers. Scribe provides a simple logging interface in several programming languages for a connection to local server and event reporting. Event in Scribe contains along the main message a category, which describes the intended destination. The server routes events based on category prefixes.

Scribe solves the application limitation problem which exists in syslogd and thus for its minimalistic approach can be seen as a complement to the syslogd in networked systems. Unfortunately, since there is no unified structure of the category and the message, it is not suitable for systems with off-the-self components where the category consistence would not guaranteed. Moreover, there is no connection for the already existing components using their proprietary logging facilities. Together with relatively poor filtering capabilities and no correlation capabilities, neither syslogd nor Scribe is suitable for our target enterprise systems.

Esper [20] is an event stream processing (ESP) and event correlation engine written in Java and C#. Esper accepts queries on temporal properties of the events in a context of sliding windows. In real time, Esper processes a stream of events and reports successful matches. Even though Esper is only a library which itself does not provide any functionality for obtaining or storing events, it is an interesting related work with respect to symptom matching. So far symptoms databases we have worked with did not contain temporary criteria, but we see the time as an important property for further system analysis and verification.

Log and Trace Analyzer [21] (LTA) has been developed as a "single point of operation" in the Eclipse environment for analysis of system logs and traces and uses the same underlining technology from the Autonomic Toolkit as the RTCE. The LTA allows importing, merging, and browsing several various log files. Symptom databases could be imported as well, which allows to associate events with symptoms, and even to add or modify entries in the database. However, the LTA has not been primarily developed for real-time operations. For example, symptom matching is processed only on "static" data and is not continuously updated. Also, the LTA keeps all the processed events in an operational memory which is not suitable for large logs.

Yemanja [22] is a correlation engine used for network problem determination and SLA [23] violation detection. Devices and components in the system are represented by a layered entity models. Each entity has defined a set of scenarios that represent a particular erroneous behaviour. A scenario contains a set of rules that con-

sume (match) some of the incoming events. When some of the rule is satisfied, the scenario is executed and a cause event (a composite event [24]) is propagated to the higher layer of entities.

Yemanja expects the events to be well-known, understood, and described. Its purpose is to aggressively filter-out as much events as possible and present only those most important. This is very different from our perspective and experience. We deal mainly with unexpected behaviour and we need to keep a relatively high degree of details for further analysis and reporting.

7. CONCLUSION

Collecting information about the software system is an important prerequisite for all self-managing, self-healing, and self-optimising systems. We have presented the Run-Time Correlation Engine, an architecture which in run-time collects events from the system, unifies into a common format, correlates, filters, and analyses against known symptoms. Automatic discovery of new component instances in the network and a resilience and automatic recovery from component and network failures makes the RTCE optimal for dynamic systems with frequent reconfiguration. Using effective data structures and algorithms, we have demonstrated an implementation which scales well for an increasing number of events and symptoms.

The RTCE can be seen as an implementation of the monitoring and analysis components of the autonomic manager, achieving Level 2-3 of autonomic maturity [4].

Acknowledgments.

We would like to thanks the following experts whose expertise and experience helps us in building the RTCE. It is Eric Labadie, Autonomic Computing, Tivoli, IBM Toronto, Wanjun Wang, SVT IBM Raleigh, and Morten Kristiansen, Mark Curran, Gary Denner, Christopher Malins, and Tracy Green from SVT IBM Dublin.

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre, and by the Enterprise Ireland Innovation Partnership in cooperation with IBM and University College Dublin. Viliam Holub and Trevor Parsons are supported by the IRCSET Embark Postdoctoral Fellowship Scheme.

8. REFERENCES

[1] J. O. Kephart and S. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[2] R. Sterritt and D. W. Bustard, "Towards an autonomic computing environment," in *Proceedings of the 14th International Workshop on Database and Expert Systems Applications*. IEEE Computer Society, 2003, pp. 694–698.

[3] B. Roehm, B. Czepregl-Horvath, P. Gao, T. Hikade, M. Holecky, T. Hyland, N. Satoh, R. Rana, and H. Wang, *IBM WebSphere V5.1 Performance, Scalability, and High Availability, WebSphere Handbook Series*. IBM Redbooks, 2004.

[4] B. Jacob, R. Lanyon-Hogg, D. K. Nadgir, and A. F. Yassin, *A Practical Guide to the IBM Autonomic Computing Toolkit*. IBM Redbooks, 2004. [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg246635.pdf

[5] "Autonomic computing toolkit." [Online]. Available: http://www.ibm.com/developerworks/autonomic/overview.html

[6] "Understanding Common base events specification v1.0.1." [Online]. Available: http://www.ibm.com/developerworks/autonomic/books/fpy0mst.htm#HDRAPPA

[7] "Web services distributed management (WSDM)," OASIS standard, 2006.

[8] G. Grabarnik, A. Salahshour, B. Subramanian, and S. Ma, "Generic adapter logging toolkit," in *ICAC'04*. IEEE Computer Society, 2004, pp. 308–309.

[9] "Eclipse test & performance tools platform project." [Online]. Available: http://www.eclipse.org/tptp/

[10] "Symptoms deep dive, Part 1: The autonomic computing symptoms format." [Online]. Available: http://www-128.ibm.com/developerworks/autonomic/library/ac-symptom1/index.html

[11] S. K. Chilukuri and K. Doraisamy, "Symptom database builder for autonomic computing," in *ICAS'06*. IEEE Computer Society, 2006, pp. 32–33.

[12] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[13] H. H. Krishna, "Create a static adapter for use with the Generic log adapter." [Online]. Available: http://www.ibm.com/developerworks/autonomic/library/ac-static/

[14] B. Subramanian, "Improve the run-time performance of the Generic log adapter, Part 1: A guide to writing efficient rule sets." [Online]. Available: http://www.ibm.com/developerworks/library/ac-savvy/

[15] C. Paniagua, F. Xhafa, and T. Daradoumis, "Gridifying IBM's Generic log adapter to speed-up the processing of log data," in *CISIS'07*. IEEE Computer Society, 2007, pp. 257–262.

[16] G. Jakobson and M. Weissman, "Alarm correlation," vol. 7, pp. 52–59, Nov 1993.

[17] M. Steinder and A. Sethi, "The present and future of event correlation: A need for end-to-end service fault localization," in *SCI-2001 V*, 2001, pp. 124–129.

[18] "Syslog-ng logging system." [Online]. Available: http://www.balabit.com/network-security/syslog-ng/

[19] "The Scribe project." [Online]. Available: http://developers.facebook.com/scribe/

[20] "Event stream intelligence with Esper and NEsper." [Online]. Available: http://esper.codehaus.org/

[21] "Overview of the Log analyzer." [Online]. Available: http://publib.boulder.ibm.com/infocenter/eruinf/v2r1m1/index.jsp?topic=/com.ibm.ac.lta.web.ui.help.doc/ref/resupplog.html

[22] K. Appleby, G. Goldszmidt, and M. Steinder, "Yemanja - A layered event correlation engine for multi-domain server farms," in *Proceedings, 2001 IEEE/IFIP International Symposium on Integrated Network Management*. IEEE Computer Society, 2001, pp. 329–344.

[23] A. N. Hiles, *Service Level Agreements: Measuring Cost and Quality in Service Relationships*. Chapman & Hall, 1993.

[24] G. Liu, A. Mok, and E. Yang, "Composite events for network event correlation," in *Integrated Network Management VI*, 1999, pp. 247–260.