Acta Cybernetica **15** (2002) 705–723.

Platform Independent Tool for Local Event Correlation

Risto Vaarandi*

Abstract

Event correlation plays a crucial role in network management systems, helping to reduce the amount of event messages and making their meaning clearer to a human operator. In early network management systems, events were correlated only at network management servers. Most modern network management systems also provide means for local event correlation at agents, in order to increase the scalability of the system and to reduce network load. Unfortunately all event correlation tools currently available are commercial, quite expensive, and highly platform dependent. The author presents a free platform independent tool called *sec* for correlating network management events locally at an agent's side.

1 Introduction

Network management systems were introduced in the middle of the 1980s, in order to reduce the management costs of wide and local area networks, servers, computer applications, and services[1]. One important goal that network management systems were designed to achieve was the automation of system monitoring. A primitive example of the monitoring automation is a shell script that executes once in every 5 minutes and uses the *ping* application to check the availability of some important servers. If a server is not responding to the *ping*, the script sends an SMS-message to the system administrator's mobile phone.

Today's network management system consists of *managed objects* and *manager objects*. A managed object is a device, a computer, or an application that requires monitoring and management. A manager object is usually a dedicated *management server* that runs specific software (such as HP OpenView [7, 8] or Tivoli [2]) to perform monitoring and management tasks on managed objects. In order to monitor or manage a particular managed object, the management server contacts an *agent* that runs at the same physical node as the managed object and acts on behalf of the management server (see Figure 1).

*Department of Computer Engineering, Tallinn Technical University, Ehitajate tee 5, Tallinn 19086, Estonia, e-mail: risto.vaarandi@eyp.ee

[1]The term *network management* has become rather generic in its nature, and it is very often used to refer to a server, application, and service management as well.

Important status changes of managed objects that are observed by agents are called *events*. Examples of events are a link loss on a router, a high load on a server, or a broken TCP connection between two applications. The management server discovers new events by polling agents or by receiving notifications from them. Discovered events are presented as *event messages* to human operators at the management server's console.

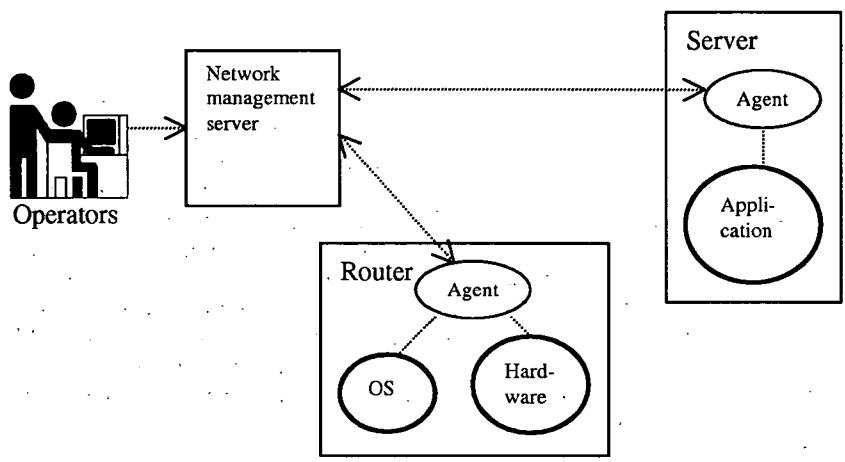

Figure 1: An example network management system (managed objects are depicted as bold circles).

If a network management system is a larger one with hundreds or thousands of managed objects, the amount of event messages often becomes too large to be handled effectively by a human. Especially undesireable situation is an *event message storm* - the flood of event messages triggered by a single hardware, software, or network failure.

In order to reduce the amount of event messages seen by the operator and to cope with various kinds of event message storms, event correlation is used. *Event correlation* is a process where irrelevant events are filtered out from being presented to a human operator and new events are derived from existing ones. For instance, events "Internal temperature of a device is too high" and "Device is unreachable" could be replaced by a single event "Device stopped working due to high internal temperature".

Most modern network management platforms provide *event correlation engines* for solving event correlation tasks [2, 6]. Since early network management systems, the network management server has been a primary location for the event correlation engine, that allows the engine to see the events of all managed objects and to correlate events even if they come from different sources.

That location of a correlation engine poses a potential threat to the scalabil-

ity of the system, since the engine processes the events of the whole system and can therefore become a bottleneck. In particular, if agents do not analyze and filter events locally and leave all that work to the correlation engine, the network management server and the engine itself could soon become overloaded. From the point of view of scalability, it would be ideal to have intelligent agents that could correlate as much events as possible locally, leaving only those events to the central correlation engine at which the knowledge of *global* context is necessary in the correlation process. Additional event correlation at an agent's side would shift load from a single network management server to many agents, reducing the possibility that the management server will become a bottleneck. Local event correlation at agents would also reduce network load, since fewer events would be reported to the network management server.

Most modern network management platforms (such as HP OpenView and Tivoli) address this problem and provide means for local event correlation [6, 7, 16].

There are two methods for correlating events locally (see Figure 2). The first method is to redirect an agent event stream (all events that go from the agent to a network management server) to a local event correlation engine, so that the network management server receives output events of the correlation engine. The other method is to correlate events *at their source*, whereby the correlation engine receives its input events from a local event source (such as application's logfile), directing its output events to the agent, while the agent event stream goes directly to the management server and remains uncorrelated. In that case the correlation engine is not located between the agent and the network management server, but between the agent and the local event source.

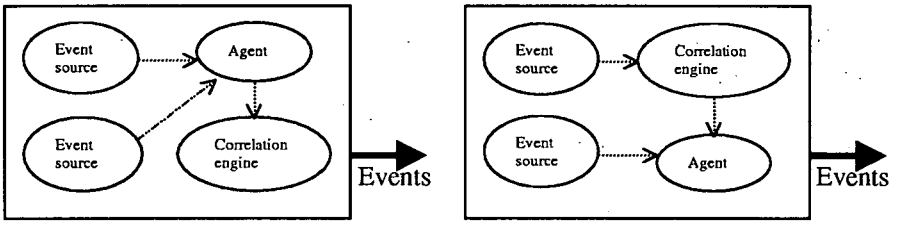

Figure 2: The methods of local event correlation.

In this paper the author presents a platform independent tool for local event correlation called *sec* (Simple Event Correlator), that implements a small set of correlation operations essential in practice. *Sec* can be used as a correlation engine for the whole agent event stream, but also for individual event sources that are files (or accessible through the file interface).

The rest of this paper is organized as follows: section 2 gives an overview of event

correlation techniques and operations; section 3 discusses related work; section 4 contains a short description of *sec*; section 5 discusses correlation rule types implemented in *sec*; section 6 describes an experiment for measuring *sec* performance; section 7 discusses the current status of *sec* and provides availability information; and section 8 concludes the paper.

2 Event correlation

Over the past ten years, many event correlation techniques have been introduced. Meira [14] provides a good overview of existing approaches.

One of the common approaches today is rule-based correlation. In the rule-based correlation, all knowledge necessary for event correlation is contained in a rule base, where each rule has the form *IF condition THEN action.* The rule-based correlation is suitable for cases where relations between events are well known and can be clearly formulated. The main disadvantage of the rule-based correlation is the lack of the learning process - past experience is not used for deriving new knowledge. This disadvantage has motivated several event correlation approaches that involve various other AI techniques, most notably neural networks. Although these techniques are quite appealing, they often suffer from the fact that their reasoning process remains unclear for the end user. For instance, neural network based computer applications sometimes produce results that are entirely unexpected even for the author of the application. However, it is essential for an AI application to reason in a manner that is clear and transparent for humans - if end users do not understand *why and how* the application reached its output, they tend to ignore the results computed by that application [15]. In the case of rule-based event correlation engines, the event correlation process is fully determined by the user-customizable rules. Since this allows the end users to fully control the work of the event correlation engine, the rule-based approach is the most accepted approach among network management engineers, and also most widely employed in today's event correlation engines [2, 4, 6, 10, 13]. In order to address the knowledge acquisition and learning problems of the rule-based approach, data mining techniques have been successfully used [11].

Jakobson and Weissman [10] provide a classification of operations that can be carried out on events during the correlation process (this classification has been adopted in a number of research papers like [11, 14]):

- Compression - substitute repeated events A with a single event A.
- Suppression - suppress an event A, if a certain operational context is present.
- Filtering - suppress an event A, if one of its parameters has a certain value.
- Counting - count repeated events A and if their number exceeds a certain threshold, replace them with a single event B.
- Scaling - in the presence of a certain operational context, replace an event A with an event B, where one of the B's parameters takes a higher value.

- Generalization - replace an event A with a more general event B, if a certain operational context is present.
- Specialization - replace an event A with a more specific event B, if a certain operational context is present.
- Temporal relationship - correlate events depending on the order of their arrival and/or the time of their generation.
- Clustering - generate an event A, if more complex correlation patterns are detected on received events. Clustering operation may take into account the result of some external tests, or combine several correlation operations listed above.

An important characteristic of an event correlation engine is its performance. Although there is no formal definition for "good performance", the event correlation engine is considered to perform well if it is able to process event floods (hundreds of input events per few seconds) in a timely manner, and if it consumes little system resources (most notably CPU time and memory). The latter condition is especially important for the local event correlation, since the local hardware resources are often quite limited (e.g., consider an event correlation task on a workstation with weak CPU and small amount of memory).

In the following section related work on event correlation is discussed.

3 Related work

Commercial network management platforms like HP OpenView and Tivoli provide network management agents that usually have some support for event correlation at event sources. The logfile monitoring module of the HP OpenView ITO agent supports compression and counting operations [7]. In the case of Tivoli Distributed Monitoring, the customer has many predefined modules that support event correlation at event sources and can develop his/her own module if desired [16].

In order to extend the capabilities of standard agents, some network management platforms provide full-featured correlation engines for agents, designed for correlating the event stream of an agent that goes to a management server. For example, HP ECS correlation engine [6] that usually runs on a network management server is also integrated into some versions of the HP OpenView ITO agent [7].

There are some obstacles to using commercial correlation engines, however. The first problem is that commercial engines are quite expensive; for instance, the price of HP ECS is around US$30,000. Many of them work only with one particular network management platform, so the customer is unable to use them independently or with other platforms.

The second problem is that commercial correlation engines have a limited support for different operating systems. For example, HP ECS is currently integrated only into HP-UX, Solaris, and Windows NT versions of the HP OpenView ITO agent. The support for different operating system platforms could be improved by

putting the source code of the correlation engine into a public domain (in order to encourage porting to other platforms), but this is not acceptable for many vendors.

Since a lot of research has been done in the field of event correlation over the past few years, some non-commercial correlation engine prototypes have been created (experimental engines presented in [5], [13], and [17] are good examples of such work). There is, however, no freeware correlation engine available yet which would be mature enough for using in a production environment. OpenNMS team that is working on the BlueBird project[2] also plans to implement the MAJI correlation engine after first versions of BlueBird have been released, but currently only the code specification of MAJI is available.

Few event correlation operations are also implemented in some freeware logfile monitoring tools. Swatch [9] supports event compression operations, allowing one to suppress repeated events in a given time frame. Logsurfer [12] is a more sophisticated tool that also supports temporal relationship operations.

One feature that almost all logfile monitors lack of is the possibility to recognize events which span over multiple logfile lines. In most logfiles each event is described by a single line, but there are still cases where one event covers two or more physical logfile lines. For example, *cron* daemons of HP-UX and Solaris write the time and the description of a single event to two consecutive lines. In addition to providing event correlation facilities for logfiles, *sec* also addresses this problem.

In the following section design goals and a short description of *sec* are presented.

4 Design goals and description of *sec*

Sec is a rule-based event correlation tool that was designed for UNIX-like operating systems. The rule-based approach was used because it is most clear and transparent for the end user (see section 2). *Sec* receives its input events from a file stream, reading new information line by line, and produces output events by executing user-specified shell commands. Design goals of *sec* were as follows:

- It should be able to handle input events regardless of their format.
- It should not be tied to any particular operating system.
- It should not be tied to any particular network management platform, but provide generic interface that makes integration with any platform possible.
- It should be possible to use it as a correlation engine for individual file-like event sources (e.g., logfiles).
- It should implement a set of correlation operations that are essential in practice, covering all operation types listed in section 2.
- It should be able to handle input events even if they arrive at a high rate.

[2] BlueBird project is an attempt to create a freeware network management platform, in order to provide a powerful and cost-effective alternative to expensive commercial platforms. See http://www.opennms.org for the current status of BlueBird and MAJI.

To be able to handle input events regardless of their format, *sec* uses regular expressions for recognizing them. Regular expressions are a natural choice because they are able to match complex patterns in input data. *Sec* also supports the use of regular expressions that match patterns spanning over multiple input lines, so input events are not restricted to have a "single line" format.

To achieve the independence from operating system platforms, the author decided to write *sec* in Perl. Perl is a widely used scripting language that runs on almost every UNIX flavour and is a standard part of many UNIX distributons. This means that applications written in Perl are able to run on a wide range of operating systems. In addition to these advantages, the support for regular expressions is integrated directly into the Perl language core. Perl programs are also almost as fast as programs written in C.

Perl 5.005 or higher is required to run *sec*, because it uses some Perl constructs for compiling regular expressions that were introduced in version 5.005. *Sec* has currently been tested on Linux, HP-UX, and Solaris, but it should run on most modern UNIX flavours.

If a regular file or standard input is specified as an input file for *sec*, *sec* acts as a correlation engine for that individual event source. In order to make the integration with arbitrary network management platform possible, sec also supports named pipes as input files. Specifying a named pipe as input is a convenient way to achieve communication between *sec* and network management agents, since the agent that supports external correlation engines can redirect its event stream from network management server to the pipe[3]. Events that the agent writes to a named pipe can have any format (and even span over multiple lines), as long as they can be matched by regular expressions.

At startup *sec* reads rules from its configuration file, opens the input and waits for new bytes to arrive. When new data become available, *sec* reads the data and updates its internal input buffer that holds N last input lines. Rules are processed then (in the same order as they were specified in the configuration file), comparing the condition part of every rule with the current content of the input buffer. After a match has been found and the action part of the rule has been executed, *sec* will optionally continue the search for new matches. The rules allow not only shell commands to be executed as actions, but they can also:

- create and delete contexts that decide whether a particular rule can be applied at the moment,
- generate new events that will serve as an input for other rules,
- reset correlation operations that are performed by other rules.

This makes it possible to combine several rules and form more complex event correlation schemes.

In the following section rule types implemented in *sec* are discussed.

[3]The author has successfully integrated *sec* with the HP OpenView ITO agent, using a named pipe for event passing.

5 Description of event correlation rule types implemented in *sec*

In the design process of *sec* the following question had to be answered - what rules should be implemented and why? Since *sec* was intended to be a tool that implements rules essential in practice, a number of event correlation guides from Cisco [1], IBM [3, 4], and HP [7, 8] were used for answering the question. Rules were derived from suggestions and example cases that were presented in more than one source document.

Rules that are currently implemented in *sec* support basic forms of *compression*, *suppression*, *filtering*, *counting*, *temporal relationship*, and *clustering* operations (see section 2). By combining several rules, many variants of every correlation operation from section 2 can be configured.

Sec configuration file consists of rule definitions, one definition per line, with whitespace-bar-whitespace string used as a field separator. Most rule definitions have the following parts:

- **Rule type** - one of the *Single*, *SingleWithScript*, *SingleWithSuppress*, *Pair*, *PairWithWindow*, *SingleWithThreshold*, *SingleWith2Thresholds*, *Suppress*, and *Calendar*.

- **Behaviour after match** - specifies whether the search for matching rules should continue after a match has been found between the current rule and input line(s). One of *TakeNext* and *DontCont* strings must be specified as a value.

- **Pattern and its type** - specifies the pattern that the input line(s) will be compared with in order to discover an event. Both regular expressions (type *RegExp*) and strings (type *SubStr*) can be used as patterns. If the type is followed by a number, the number specifies how many input lines will be used in comparison.

- **Event description** - textual description of the discovered event.

- **Action** - the action that will be executed when an event has been discovered. Actions include executing a shell command, creating a context, deleting a context, and resetting a correlation operation (e.g., reset the counting operation that is currently performed by some other rule).

- **Counting and timing constraints** - optional constraints for implementing correlation operations like counting and temporal relationship.

- **Context** - the optional context where the rule is considered valid at runtime.

In the following sections *sec* rule types are described.

5.1 Single rule

This rule does not implement any of the correlation operations but takes immediate action if certain line(s) appear in the input. It can be used as a building block for creating more complex event correlation operations. Here are some example definitions of this rule:

```
Single | DontCont | SubStr | start of maintenance |
maintenance | create

Single | DontCont | SubStr | end of maintenance |
maintenance | delete

Single | DontCont | RegExp2 | ^database error:\n(.*) |
DB error: $1 | shellcmd event.sh "%s" | !maintenance
```

The first rule creates the context *maintenance* if a line containing the substring *"start of maintenance"* appeared in the input. The second rule deletes that context if a line containing the substring *"end of maintenance"* was observed. The third rule generates an output event by executing

```
event.sh "DB error:  <error description>"
```

(%s is replaced by the event description), if a database error occurs and the application is currently not maintained (this example assumes that two consecutive lines appear in the input in the case of a database error - the line *"database error:"* followed by a line with a detailed error description).

5.2 SingleWithScript rule

This rule was designed to integrate external programs with *sec* event flow, implementing a form of the *clustering* operation. If a matching event appears in the input, an external program given by the rule definition is executed, and if the program returns zero for its exit value, then an action is executed. The definition of this rule is identical to the definition of the *Single* rule, except of the additional parameter that specifies an external program.

5.3 SingleWithSuppress rule

This rule was designed to implement one of the basic forms of the *compression* operation. If an event A is observed, the rule executes an action immediately but ignores all other instances of the event A that will appear during next t seconds. This operation is commonly needed in practice - it is implemented, for example, in HP OpenView Network Node Manager as one of the basic correlation schemes [8]. It is also provided as an example correlation scheme in Tivoli manuals [3]. In addition, rules with the same semantics are supported by logfile monitoring tools like Swatch [9] or the logfile encapsulator of HP OpenView ITO [7].

If a single hardware, software or network failure causes hundreds of events to appear in the input, this rule is useful for acting on the first matching event and ignoring all the other for a given time period. For example, if a file system becomes full, then every attempt to write to a file in that file system causes *"file system full"* message to be logged with *syslog* in many UNIX environments. If the file system that was filled up is used extensively, thousands of identical lines may be written to a logfile within few seconds, while only the first of them is important and all the other redundant[4].

Here is an example rule for handling these messages in HP-UX, that compresses all *"file system full"* events from a logfile into a single event (compression takes place for 15 minutes), and notifies local agent by calling *notify.sh* script:

```
SingleWithSuppress | DontCont | RegExp |
(\S+) [fF]ile system full | File system $1 full |
shellcmd notify.sh "%s" | 900
```

Note that *sec* considers events identical only if their descriptions are identical, so the event *"File system /home full"* will not be suppressed if it appears after *"File system /usr full"* event.

5.4 Pair rule

This rule was designed to implement one of the most common forms of the *temporal relationship* operation. If an event A is observed, the rule executes an action but ignores all other instances of the event A as long as an event B has not been observed. When the event B is observed, the rule will complete its work by executing another action. Like the previous operation, it is implemented in HP OpenView Network Node Manager as one of the basic correlation schemes [8]. It is also provided as an example in Tivoli manuals [3, 4].

This rule is useful for reducing two or more events into an event pair. A good example of application of this rule is NFS monitoring. When the NFS server becomes unreachable, the NFS client starts to log *"NFS server not responding"* messages. When the NFS server becomes reachable again, a single *"NFS server ok"* message is logged. Here is an example rule for events from a Solaris NFS client logfile that ignores redundant error events for 1 hour, producing an event both for the start and for the end of an error condition:

```
Pair | DontCont | RegExp | NFS server (\S+) not responding |
$1 is not responding | shellcmd notify.sh "%s" |
SubStr | NFS server $1 ok | $1 OK | shellcmd notify.sh "%s" | 3600
```

[4]Some implementations of syslog daemon cope with this and similar situations by implementing event compression internally.

5.5 PairWithWindow rule

This rule was designed to implement another variant of the *temporal relationship* operation. If an event A is observed, the rule waits for t seconds to see if an event B also appears (all subsequent instances of A are ignored during the waiting). If the event B is not observed within t seconds, then the action corresponding to the event A is executed; if the event B arrives on time, the action corresponding to the event B is executed.

This rule is needed when an error event becomes relevant if the error is not cleared after a certain amount of time. This is often the case for network events - in Cisco event correlation guide several common network management scenarios with Cisco devices are provided, where the presence of this rule is required for event correlation [1]. It is also provided as an example correlation scheme in Tivoli manuals [3].

Here is an example rule for events from a Cisco router logfile (the router must have *syslog* message logging enabled) that detects situations where the router does not come up within 5 minutes after an administrator has rebooted it with the *reload* command:

```
PairWithWindow | DontCont | RegExp |
(\S+) \d+: %SYS-5-RELOAD | $1 did not come up after reboot |
shellcmd notify.sh "%s" | RegExp | ($1) \d+: %SYS-5-RESTART |
$1 successful reboot | shellcmd notify.sh "%s" | 300
```

5.6 SingleWithThreshold rule

This rule was designed to implement the *counting* operation. The rule counts instances of an event A during t seconds, and if the number of events becomes equal to the threshold n before t seconds have elapsed, the rule executes an action. All subsequent instances of the event will be ignored for the rest of the time window. The time window is sliding - if less than n but more than 1 events were observed during t seconds, the beginning of the window is moved to the time moment when the second event took place.

Like the previous rule, this rule is often needed in correlating network events [1]. A rule with similar semantics is also supported by the logfile encapsulator of HP OpenView ITO [7].

This rule is useful when an event must occur repeatedly to become relevant, and a single instance of that event can be ignored. Here is an example rule for events from a Solaris *bad logins* logfile that executes *notify.sh* script, if three login failures for the same user on the same terminal were observed within 1 minute:

```
SingleWithThreshold | DontCont | RegExp | (\S+):(\S+): |
Repeated login failures for user $1 at tty $2 |
shellcmd notify.sh "%s" | 60 | 3
```

5.7 SingleWith2Thresholds rule

This rule was designed to implement another variant of the *counting* operation. The rule counts instances of an event A during t seconds and executes an action if a given threshold n is exceeded (exactly like the previous rule does). What is different from the previous rule is that the counting continues after the action has been executed - if no more than n' events A will be observed during t' seconds, the rule will execute another action. Both time windows are sliding.

This rule is useful when both the start and the end of the error condition can be discovered by counting. Here is an example rule for events from a Cisco router logfile that detects CPU overload conditions (two SYS-3-CPUHOG messages are logged within 1 minute) and also generates an event when CPU load is normal again (no SYS-3-CPUHOG messages are observed during 15 minutes):

```
SingleWith2Thresholds | DontCont | RegExp |
(\S+) \d+: %SYS-3-CPUHOG | $1 CPU overload |
shellcmd notify.sh "%s" | 60 | 2 | $1 CPU load normal |
shellcmd notify.sh "%s" | 900 | 0
```

5.8 Suppress rule

This rule was designed to implement *suppression* and *filtering* operations. Here is an example of the suppression operation (if the context *mycontext* is present, suppress all *"file system full"* events):

```
Suppress | RegExp | (\S+) [fF]ile system full | mycontext
```

Here is an example of the filtering operation (all file systems belonging to the volume group *vg01* are filtered out from being reported as full):

```
Suppress | RegExp | /dev/vg01/(\S+) [fF]ile system full
```

5.9 Calendar rule

This rule was designed for executing actions at specific times. Unlike all other rules, this rule reacts only to the system clock, ignoring other input. Time moments when the rule must act are specified in *crontab*-style. This rule can be used as a building block for creating more complex time-related event correlation operations, or for other time-related purposes.

The following example rule creates the context *NightContext* every day at 11PM; the context has a lifetime of 9 hours:

```
Calendar | 0 23 * * * | NightContext | create 32400 %s
```

5.10 Other variants of correlation operations

The rules that were described in the previous sections implemented some basic forms of *compression*, *suppression*, *filtering*, *counting*, *temporal relationship*, and *clustering* correlation operations. Other variants of correlation operations can be configured by combining several rules with appropriate actions.

Here is an example of the *specialization* operation that generates an error event if the application failed to process three subsequent incoming queries. If this was caused by a high CPU load, a more specific error event is generated.

```
# Create the context "cpu_overload" if the CPU load is
# too high, and delete it if the load is normal again

Pair | DontCont | SubStr | CPU load is too high |
cpu_overload | create | SubStr | CPU load is normal |
cpu_overload | delete | 86400

# Count the number of failed queries, and generate an
# event "3 subsequent failed queries" if the threshold
# is exceeded

SingleWithThreshold | DontCont | SubStr |
query processing failed | 3 subsequent failed queries |
event | 600 | 3

# Reset the counting done by the previous rule if some
# query is processed successfully (event text "3
# subsequent failed queries" is used to refer to the
# pending counting operation)

Single | DontCont | SubStr | query processing successful |
3 subsequent failed queries | reset

# Generate a more specific error event for output

Single | DontCont | SubStr | 3 subsequent failed queries |
Three subsequent queries have failed due to high CPU load |
shellcmd sendevent.sh "%s" | cpu_overload

# Generate a general error event for output

Single | DontCont | SubStr | 3 subsequent failed queries |
Three subsequent queries have failed |
shellcmd sendevent.sh "%s"
```

The following section describes an experiment for measuring *sec* performance.

6 Performance of *sec*

This section describes an experiment for measuring *sec* performance that was conducted in the Union Bank of Estonia. The experiment lasted for one week. Performance measurements were obtained from the computer that was running *sec* to monitor the banking card service.

The purpose of the card service is to offer the clients an opportunity to use banking cards for payment and for cash withdrawal. Card service consists of the following components - automatic teller machines (ATMs), point of sale terminals (POS terminals), and a card server. An ATM is a device that is primarily used for cash withdrawal by the clients. A POS terminal is a device that merchants rent from the bank, so that their customers can use banking cards for payment. The card server is a service process that receives requests from ATMs and POS terminals and processes them if the client who issued the request is authorized for it.

The card server keeps a detailed log about card service events (client requests, card server answers, card server internal errors, etc.), that served as an input for *sec*. During the experiment, new events arrived at a high rate - during daytime non-peak hours, about 3000-4000 new events (ca 300-400 KB of new information) were appended to the log every minute. At peak hours (between 4.00 and 6.00 PM), the arrival rate was 5000-6000 events (ca 500-600 KB) per minute.

Due to the high arrival rate of events, it was decided to put *sec* between the event source and the agent (and not between the agent and the network management server), in order to correlate events as early as possible. For security reasons, it was also decided that *sec* must not run on the card server machine, but on a separate computer and receive the card server log through a named pipe (card server log was sent over the network and written to the pipe by a separate shell script; see Figure 3).

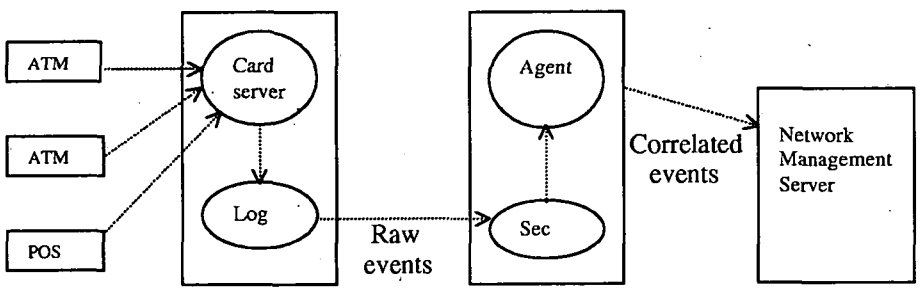

Figure 3: The experiment for measuring *sec* performance.

A low-end desktop computer with 200MHz Intel Pentium processor[5], 32Mb of memory, and Linux as an operating system was used for running *sec*. Since HP OpenView is used as the network management platform in the Union Bank of Estonia, the HP OpenView ITO agent was also installed on that machine. *Sec* was configured to use *opcmsg* utility for producing output events (*opcmsg* is an HP OpenView tool for generating events that are received by the local ITO agent). There were 63 rules specified in the configuration file of *sec*: 29 *Single* rules, 9 *Pair* rules, 1 *PairWithWindow* rule, 5 *SingleWithThreshold* rules, 17 *SingleWith2Threshold* rules, and 2 *Suppress* rules.

Figures 4-7 display data about CPU utilization and 1-minute load average of the computer that was running *sec* during the experiment[6].

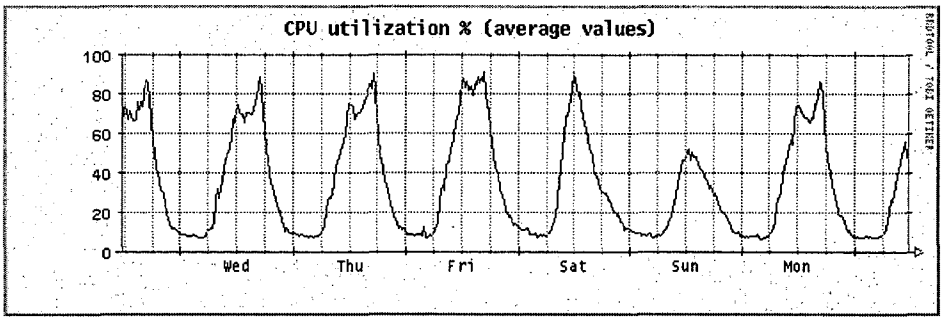

Figure 4: CPU utilization (average values).

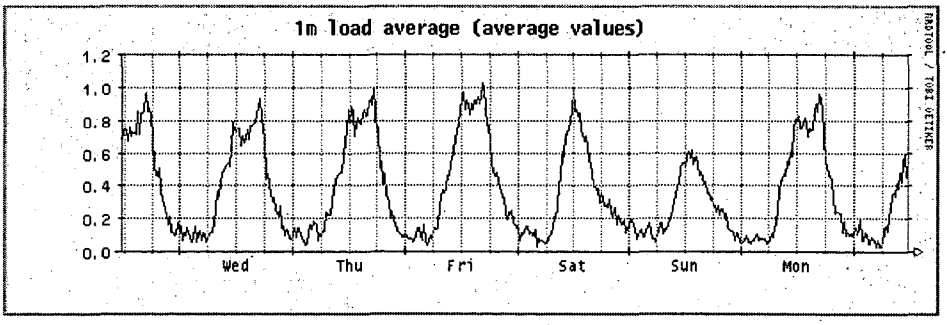

Figure 5: 1-minute load average (average values).

Graph data were gathered at 1 minute intervals. Since it was impossible to accommodate all data points to the graphs (due to their limited size), every line

[5]The processor did not have MMX support.

[6]Graphs were produced with RRDtool by Tobias Oetiker.

pixel represents the average value for 17 minute period in Figure 4 and 5, and the maximum value for 17 minute period in Figure 6 and 7.

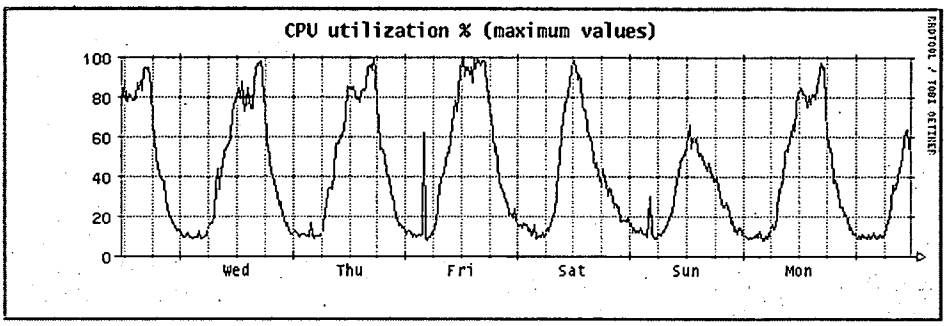

Figure 6: CPU utilization (maximum values).

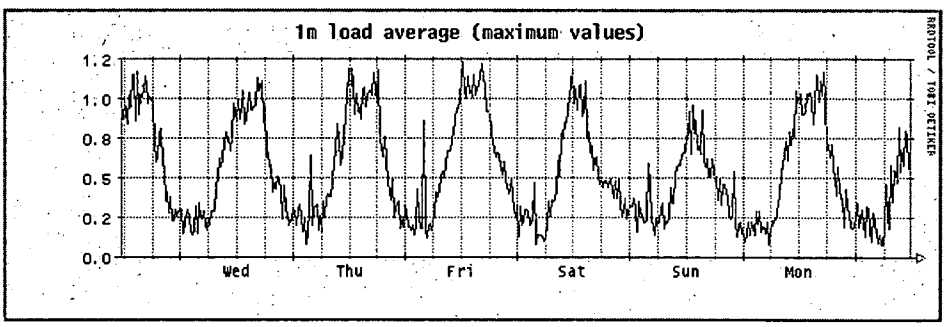

Figure 7: 1-minute load average (maximum values).

About 2.5 million events were matched by the rules during the experiment (that makes about 4 matched events per second as an average). Those input events were reduced to 101 output events by *sec*.

Performance data that were gathered shows that *sec* performs well under heavy event load. Though low-end hardware was used for conducting the experiment, *sec* was able to process input events in a timely manner without imposing heavy load on the local hardware resources. During the experiment, *sec* consumed only 4MB of physical memory. Despite the high arrival rate of events, 1-minute load average exceeded the level of 1.0 only at peak hours. Although the computer CPU was relatively slow, it was completely utilized only occasionally - in Figure 6, there are only 17 data points with the value of 100 per cent.

7 Current status of *sec* and availability

One year of experience with *sec* has proved that it is an efficient tool for correlating massive event streams locally. *Sec* has been successfully applied for network management in a number of companies in Europe and U.S., mainly in telecom and financial institutions.

It is difficult to estimate the total number of companies and organizations which are using *sec*, since no registration is required to download it. However, at the time of writing this, the *sec* download web-page had been visited more than 3,000 times. The author has received reports of successful use of *sec* on Solaris, HP-UX, Linux, and Windows2000 platforms.

In October 2001, the first version of *sec-2.0* was released, that implements a number of new action types, augments the properties of a context, and supports enchanced configuration file syntax.

Sec is distributed under the terms of GNU General Public License, and can be downloaded from *http://kodu.neti.ee/~risto/sec/*.

8 Conclusion

Although commercial network management platforms provide means for event correlation, there are still some problems that hinder the use of commercial event correlation engines - commercial engines are quite expensive, many of them do not work independently or with other network management platforms, and they also have a limited support for different operating system platforms. Since a lot of research has been done in the field of event correlation over the past few years, some non-commercial correlation engine prototypes have been created. There is, however, no freeware correlation engine available yet which would be mature enough for using in a production environment.

In this paper the author presented a free platform independent tool for local event correlation called *sec* (Simple Event Correlator), that implements a set of correlation operations essential in practice. *Sec* can be used as a correlation engine for the whole agent event stream, but also for individual file-like event sources.

For a future work, the author plans to use *sec* in other research areas which are similar to network management and which could benefit from event correlation techniques (e.g., intrusion detection).

Acknowledgments

Author wishes to thank the Union Bank of Estonia for supporting this work. Author also thanks Prof. Ahto Kalja for providing remarks and suggestions that helped to improve the quality of this paper.

References

[1] Cisco Systems. *Cisco Network Monitoring and Event Correlation Guidelines.* Reference Guide, Cisco Systems Inc., 1999.

[2] Catherine Cook, Budi Darmawan, Mike Foster, Stephane Gillardo, Vasfi Gucer, David Kong, Dinesh Kumar, Edson Manoel, Fred Plassman, Roger Reynolds, Kenshoh Sugitani, Samson Yiu. *An Introduction to Tivoli Enterprise.* Tivoli Redbook SG24-5494-00, IBM Corp., 1999.

[3] Paul Fearn, Raj Chityal, Nancy Jarin, Elise Kushner, Gordon Lilly, Darren Pike. *TEC Implementation Examples.* Tivoli Redbook SG24-5217-00, IBM Corp., 1998.

[4] Paul Fearn, Arne Olsson, Larry Bajuk, David Edwards, Peter Glasmacher, Gareth Holl, Istvan Szarka. *Integrated Network Management Solutions Using NetView Version 5.1.* Tivoli Redbook SG24-5285-00, IBM Corp., 1999.

[5] Boris Gruschke. *Integrated Event Management: Event Correlation using Dependency Graphs.* Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, 1998.

[6] Hewlett-Packard. *Event Correlation Services - Designer's Guide.* HP document J1095-90304, Hewlett-Packard Company, 1998.

[7] Hewlett-Packard. *HP OpenView IT/Operations Concepts Guide.* HP document B6941-90002, Hewlett-Packard Company, 1999.

[8] Hewlett-Packard. *Managing Your Network with HP OpenView Network Node Manager.* HP document J1240-90021, Hewlett-Packard Company, 1999.

[9] Stephen E. Hansen and E. Todd Atkins. *Automated System Monitoring and Notification With Swatch.* Proceedings of USENIX 7th System Administration Conference, 1993.

[10] G. Jakobson and M. Weissman. *Real-time telecommunication network management: Extending event correlation with temporal constraints.* Integrated Network Management IV, 1995.

[11] Mika Klemettinen. *A Knowledge Discovery Methodology for Telecommunication Network Alarm Databases.* PhD Thesis, University of Helsinki, Finland, 1999.

[12] Wolfgang Ley and Uwe Ellerman. *logsurfer(1) and logsurfer.conf(4) manual pages.* See http://www.cert.dfn.de/eng/logsurf/

[13] G. Liu, A. K. Mok, E. J. Yang. *Composite Events for Network Event Correlation.* Proceedings of the 6th IFIP/IEEE International Symposium on Integrated Network Management, 1999.

[14] Dilmar Malheiros Meira. *A Model For Alarm Correlation in Telecommunication Networks.* PhD Thesis, Federal University of Minas Gerais, Brazil, 1997.

[15] Elaine Rich and Kevin Knight. *Artificial Intelligence.* 2nd ed., McGraw-Hill, New York, 1991.

[16] Stefan Uelpenich, Robi Banerjee, Peter Holm, Alain Queffelec. *Creating Custom Monitors for Tivoli Distributed Monitoring.* Tivoli Redbook SG24-5211-00, IBM Corp., 1998.

[17] Hermann Wietgrefe, Klaus-Dieter Tuchs, Klaus Jobmann, Guido Carls, Peter Froehlich, Wolfgang Nejdl, Sebastian Steinfeld. *Using Neural Networks for Alarm Correlation in Cellular Phone Networks.* International Workshop on Applications of Neural Networks in Telecommunications, 1997.