

Log Analysis and Event Correlation Using Variable Temporal Event Correlator (VTEC)

Paul Krizak - Advanced Micro Devices, Inc.
paul.krizak@amd.com

ABSTRACT

System administrators have utilized log analysis for decades to monitor and automate their environments. As compute environments grow, and the scope and volume of the logs increase, it becomes more difficult to get timely, useful data and appropriate triggers for enabling automation using traditional tools like Swatch. Cloud computing is intensifying this problem as the number of systems in datacenters increases dramatically. To address these problems at AMD, we developed a tool we call the Variable Temporal Event Correlator, or VTEC.

VTEC has unique design features, such as inherent multi-threaded/multi-process design, a flexible and extensible programming interface, built-in job queuing, and a novel method for storing and describing temporal information about events, that well suit it for quickly and efficiently handling a broad range of event correlation tasks in real-time. These features also enable VTEC to scale to tens of gigabytes of log data processed per day. This paper describes the architecture, use, and efficacy of this tool, which has been in production at AMD for more than four years.

Tags: security, case study, syslog, log analysis, event correlation, temporal variables

1 Introduction

Log analysis is a critical component for effective automation of large cloud computing environments. As clouds grow, day-to-day operational tasks such as failing hardware become an increasing burden for datacenter operational staff. In addition, emergent behavior in large clouds causes unusual problems that often are difficult to diagnose. These issues require more complex automation techniques in system maintenance and service operation. Modern solutions such as SEC [1,2] and Splunk [3] have done a great job at scaling to large log volumes and making complex correlations feasible, but they have drawbacks. This paper presents an alternative solution we developed at AMD called the Variable Temporal Event Correlator, or VTEC.

AMD designed VTEC with multi-core and multi-system scalability in mind. Virtually every component is multi-process and/or multi-threaded to take advantage of every available CPU cycle on the system. If needed, each component can be isolated on its own machine to distribute load.

VTEC also introduces a novel method for representing temporal event data; these constructs are called *temporal variables*. Temporal variables are constructed to represent temporal data about events, such as frequency and rate of change, in a way that is immediately useful when building event-correlation rules. These rules can make use of the temporal data without the need for extra processing in the rule itself.

Finally, VTEC includes a built-in job scheduler that allows for categorization, scheduling, and prioritization of actions generated in response to events. This gives the user finer control over the sequencing and priorities of actions generated by log analysis than available before.

This paper is organized as follows: Section 2 describes the computing environment and log analysis needs that drove the creation of VTEC. Section 3 describes the internal architecture of VTEC. Section 4 details several example "rule engines" that demonstrate how VTEC can be used to correlate various kinds of events in a computing environment. Section 5 briefly describes

some of the useful rule engines implemented at AMD. Section 6 discusses designing for performance and scaling as log traffic increases. Section 7 describes the challenges currently faced using VTEC, and areas for improvement.

2 Background

As compute environments continue to grow in size, it becomes increasingly challenging to keep track of the various events happening in the grid. How does a systems team note and track the failure of a hard disk or memory? Or when one of thousands of compute nodes suffers a kernel panic? An even more complex task is to deal with the inevitable emergent behavior of a massive network of computers. Seemingly innocuous changes to systems (e.g., adding a simple cronjob to grid nodes), can have unexpected consequences (e.g., overloading NIS/LDAP servers).

In this large, ever-changing, and complex computing environment, many organizations (including AMD) have turned to the practice of autonomic computing [4] to reduce the effort that sysadmins must exert to keep the environment stable. There is a system and OS configuration aspect to this, in which tools such as Cfen-gine [5] can enable autonomic behavior. There is still a gap, however, when it comes to detecting anomalous or interesting events, correlating them, and taking automatic action to alert people or correct the problem.

In the context of large Linux/UNIX compute grids, raw event data is generally available via syslog. Standard syslog daemons, as well as more advanced ones such as syslog-ng [6], are able to send log data to a central server. Thus, collecting enough raw data to analyze is rarely difficult. However, the volume of this data is often a problem: so much data is collected that it becomes difficult to parse and extract useful information from it.

During the past several years, a number of log parsing and event correlation tools have been developed. Some Linux distributions, such as Fedora, ship with the Logwatch [8] utility installed. Logwatch parses system log files regularly and provides useful, human-readable reports via e-mail. When using Logwatch, however, sysadmins are required to wade through e-mails and make event correlations manually, so it does not scale beyond a handful of servers.

One of the first automated log analysis tools used at AMD was Swatch [7]. Swatch is very much a reactionary log analysis system. Events matching a particular regular expression can trigger events, such as an e-mail to an administrator. At AMD, rudimentary flood prevention routines often caused important events to be missed while a rule was stalled waiting for a flood timer to expire. In addition, Swatch is single-threaded, and so was unable to scale to AMD's ever-increasing log volume (about 10GB/day at a typical site).

Around 2006, as AMD's compute grids were growing at a rapid rate, the company determined the aging Swatch installation was no longer effective and chartered a project to implement a replacement log monitoring and analysis system. The parameters for the project were:

- Scale to tens of gigabytes of log data per day
- Take advantage of multiple processors (AMD's strategy is to embrace multi-core computing)
- Be able to correlate events across thousands of systems in real-time (no batch processing)
- Be able to prioritize and queue system repair jobs and e-mail alerts
- Prevent floods of alerts without missing important events
- Correlate events on arbitrary log streams (e.g. FlexLM license daemon logs)
- Ensure correlation rules are easy to read, modify, and create

At the time, SEC [1] and Splunk [3] were popular choices as core components to achieve the goals. AMD tested both thoroughly, and ultimately decided a home-grown system would be best. SEC was (and still is) a very powerful and flexible tool, but the learning curve for writing rules its rules is quite steep. This was demonstrated by [1], which was actually supposed to "demystify" SEC. However, even moderately complex SEC rules were deemed unintelligible by sysadmins unfamiliar with its use.

Splunk did not have an indexing system robust enough to handle the volume of log traffic we expected to run through it. After routing just a few hundred megabytes of syslog data through Splunk, the indexer would stop working properly despite several weeks of tuning with the help of Splunk's developers. Finally, the event correlation features were limited to Swatch-like functionality (this was in 2006: Splunk v1.0).

With its seven design goals in mind, AMD created VTEC from scratch. In the process, we evolved a novel set of data types and data handling methods that have since greatly simplified event correlation and log analysis at AMD.

3 Architecture

VTEC Components

The VTEC system consists of five modules (Figure 1). Each module has a specific, well-defined task to perform, and communicates with the other modules in a

well-defined language over TCP sockets. Splitting the system into multiple components generates a number of benefits:

- Bugs/errors in one rule engine generally do not affect the others
- Multiple processes can leverage CPU capacity in multi-core systems
- Standard interfaces between modules simplify the task of tuning modules to optimize performance or add features

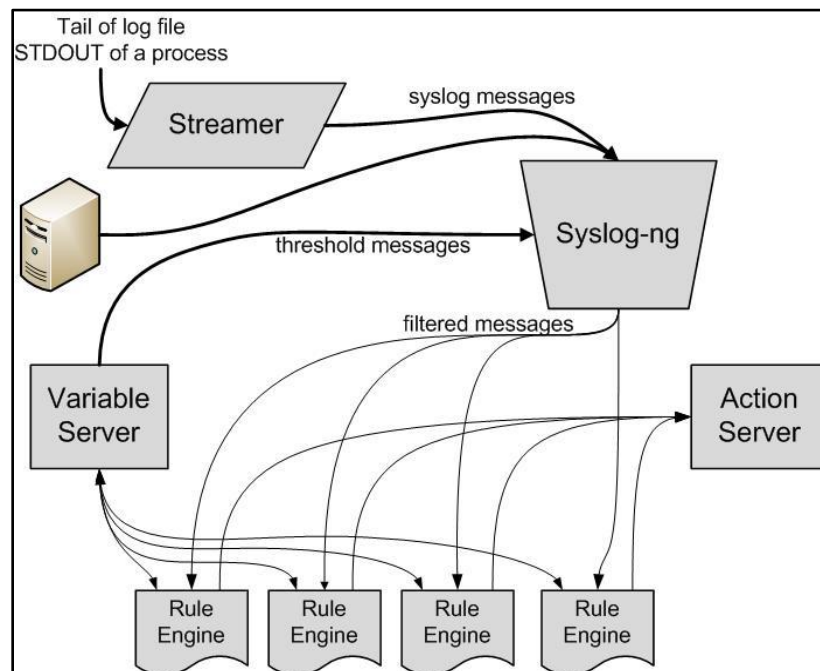


Figure 1: VTEC System Architecture

Streamer - Log data enters VTEC via syslog. The streamer component can tail log files or send arbitrary data from STDOUT of a process to VTEC. Its most useful feature is the ability to insert "heartbeat" messages into the log stream so the absence of event data can trigger actions [10]. Use of the streamer component is optional; most systems at AMD simply route their log data directly to the syslog-ng component without going through a streamer.

Syslog-ng - VTEC uses the powerful syslog-ng system logger [6] as the log router; it is the only non-Perl component. Its purpose is to accept log streams from syslog, TCP/UDP sockets, and streamers. It reformats messages, filters data, and routes messages to the

appropriate rule engines and/or archival log files. Control of the filtering is accomplished by including filtering metadata within each rule engine, dictating what log data that rule engine wishes to receive. The VTEC installer then injects this metadata into the `syslog-ng.conf` file, ensuring that each rule engine is tasked with parsing only the log data that is relevant to it. The static parts of the `syslog-ng.conf` file allow for searchable log archives to be created. For example, all log messages could go to `/var/log/YYYY/MM/DD/hostname-msgs.log`.

Rule engines - These are composed of any executable code that accepts filtered and reformatted log data on STDIN. In practice, these are Perl scripts, created from

a template, that include reusable interface modules to both the action server and the temporal variable server (which communicate over a standard TCP protocol). The rule engines are where the event correlation tasks occur. Since the rule engines are open-ended editable custom scripts, a rule engine can do anything your language of choice can do.

Temporal variable server - VTEC hosts all the temporal variables in a separate server daemon. This frees the rule engines from the drudgery of maintaining state across reboots/restarts. It also allows rule engines to share data easily, since the variable server presents a shared namespace to the rule engines; one rule engine can set a temporal variable, and another rule engine can query for it by using the same variable name.

The temporal variable server can also inject special messages into the log stream when certain threshold conditions are met. Rule engines can watch for these threshold messages and take appropriate action without being burdened with having to constantly query the state of variables.

A Perl object interface to the temporal variable server is provided for use in rule engines. Additionally, in cases in which the temporal variable server's features are more than are required, the temporal variable data types are available as Perl objects, meaning that rule engines can instantiate them locally without having to contact the temporal variable server at all.

Action server - When rule engines need to take some sort of action, they have the option of running that task locally (which is not advisable, since this can block the rule engine from processing incoming data) or queuing a job in the action server. Jobs are implemented in a Perl module as subroutines; queuing a job really means sending the name of a subroutine, its parameters, when to run it, and a queue name to the action server over a standard TCP socket interface. A Perl object interface to the action server is provided for use in rule engines.

The action server has a number of job queues with varying priorities. Users can schedule jobs to run immediately or at a specific time (e.g., alert a sysadmin about this event, but not until 10 a.m., when they are awake). The action server processes queues with higher priority first, allowing an emergency page to go out immediately despite a backlog of less urgent repair jobs in lower-priority queues.

The actions that can be queued on the action server are defined in a Perl module as functions. This allows actions to be developed that can be shared by rule engines. Since the actions are implemented in Perl, they can do virtually anything to the environment. They can also communicate with the variable server if needed to get information about how to execute. Some basic actions implemented on AMD's action server are:

- **run_cmd** - executes a command on a remote machine
- **send_mail** - sends an e-mail
- **hopenclose** - instructs AMD's batch scheduling system to open or close a host to jobs
- **inject_log** - injects a syslog message into the log stream (for signaling rule engines about job status)

In addition to the actions themselves, an arbitrary number of queues that can be defined in the action server, with varying levels of priority. The queues are defined in a configuration file. In practice, only three queues are needed:

- **crisis** - when any jobs are placed in this queue, all other queues are halted until this queue is empty (good for hotpage events or status checks that are time-sensitive)
- **normal** - normal jobs, such as rebooting or reinstalling a server, or running a script
- **email** - most e-mail jobs get queued here, so that they get processed in parallel with other jobs (e.g. a `run_cmd` job in the `normal` queue won't have to wait for a flood of 500 e-mails to get sent before executing)

Temporal Variable Data Types

The most interesting and novel aspect of VTEC is the temporal variable server and the temporal variables it hosts. There are three data types in the temporal variable server:

Scalar - A scalar consists of two pieces of information: a piece of scalar *data* (such as a string or a number) and a *timeout*. The timeout is set when the scalar is created, and defines the length of time the data is valid. When the scalar is queried, the timeout is checked. If the current time has not exceeded the timeout, the data value is returned. Otherwise, a zero is returned. Scalars are useful for setting alarms, preventing e-mail floods, and storing temporary data.

STDIN are messages from host amdftp, where the message portion matches the regular expression `disk full error`. By performing initial filtering in syslog-ng before the messages even reach the rule engine, the filtering process is only done once, rather than all rule engines having to parse through all the messages looking for matches.

When these messages arrive, the variable `server` is queried to see if a scalar by the name of `ftp_disk_full` is set. If not, an outgoing e-mail is queued on the action

server, and the `ftp_disk_full` scalar is set, with a one-hour timeout, to prevent a flood of e-mail from being sent.

Notice that the job submission hash includes a `start` parameter. This parameter can be set to a delay (e.g. `+600` means "10 minutes from now") or to an absolute time by using `localtime()` to generate an epoch time. Thus with very little added effort, a non-critical alert could be queued and delivered during business hours.

Code Example 1: Simple Event Alert with Repeat Elimination/Compression

```
# syslog-ng metadata
# filter: host("amdftp") and match("disk full error");

while(<STDIN>) {
    if($variable_server->get_scalar("ftp_disk_full") == 0) {
        my %job = (
            start      => time(),
            queue       => "email",
            action      => send_mail,
            parameters => [ $address, $subject, $msg, $name ]
        );
        if($action_server->add(%job)) {
            # scalars can be set to arbitrary values (second parameter), but
            # always return 0 when they have timed out.
            $variable_server->set_scalar("ftp_disk_full", 1, "+3600");
        }
    }
}
```

In **Code Example 2**, a more complex event correlation is demonstrated. Following [1], this code example demonstrates the functionality in both "Report on Analysis of Event Contents" and "Detect Identical Events Occurring Across Multiple Hosts". In this example, VTEC is configured to watch for "NFS server not responding" messages from all hosts, and will alert if the number of unique systems reporting this message for a particular file server exceeds 50 in a given five-minute period.

First, notice the `syslog-ng` metadata is a bit more involved this time. We are filtering three types of messages into the rule engine: `server XXX not responding`, `server XXX OK`, and `threshold` messages. The `threshold` messages are important: when the rule engine runs the `set_list_threshold` method when it starts up, the variable `server` sets up a watch for that particular list. In this case, we're telling it to watch two lists, `filer_down` and `filer_up`, and to send an alert if the value of a single key in either of them exceeds 50.

The rule engine can then capture that alert and take action.

Following the code, as log messages flow in from STDIN, they are checked to see if they are one of the three types we are filtering for. In the basic filer down/up case, the variable `server` is told to update the appropriate list variable, using the affected file server's name as the key. The timeout for each hit is set to five minutes. Conceptually, this means we'll have two groups of incrementers, indexed by the name of the file server. The value of each incrementer tells us roughly how many hosts are simultaneously reporting the given state for that server (we assume each host only reports the "not responding" or "OK" message once every five or more minutes). If the message is a `threshold` message, all we need to do is take action - in this case, send an e-mail about the affected filer.

The important thing to grasp with this example is the relative simplicity with which this correlation was achieved, especially compared to the same example in Figure 9 of [1]. While the overall functionality is effec-

tively the same, any sysadmin able to read basic Perl should be able to interpret, modify, and use the VTEC rule engine; the same functionality in SEC is much more difficult to grasp, even for a seasoned programmer.

Code Example 2: Correlating Events Across Hosts

```
# syslog-ng metadata
# filter: match(".*server.*not responding.*") or \
#         match(".*server.*OK.*") or \
#         filter(f_thresholds);

# Set up a threshold when we start up. Thresholds are purged after 48 hours of inactivity.
# VTEC restarts all rule engines nightly (during log rotation) to ensure needed thresholds
# are kept fresh, and retired ones are purged from the variable server.
$variable_server->set_list_threshold("ONE", "filer_down", ">", "50");
$variable_server->set_list_threshold("ONE", "filer_up", ">", "50");

my $window_secs = 300; # sliding window of 5 minutes
my $email_flood = 1800; # seconds between e-mails about a filer up/down

while(<STDIN>) {
    my %message = parse($_);

    # Filer down messages
    if($message{message} =~ /server (\w+) ( is)? not responding/) {
        $variable_server->set_list("filer_down", $1, "+$window_secs");
        next;
    }

    # Filer up messages
    if($message{message} =~ /server (\w+) OK/) {
        $variable_server->set_list("filer_up", $1, "+$window_secs");
        next;
    }

    # Filer up/down threshold exceeded messages
    if($message{message} =~ /THRESHOLD EXCEEDED.*filer_(up|down).*\[(\w+)\].*\((\d+)\)/) {
        my ($type, $filer, $num_messages) = ($1, $2, $3);
        # Create a scalar that we'll use to prevent e-mail flooding for this filer
        my $scalar_name = "email_filer_$type" . "_$filer";
        unless($variable_server->get_scalar($scalar_name)) {
            my %job = (
                start    => time(),
                queue     => "email",
                action    => "send_mail",
                parameters => [ $to_address, "Filer $filer being reported $type!",
                    "Filer $filer has been reported $type $num_messages times in the last $window minutes." ]
            );
            Queue(\%job);
            $variable_server->set_scalar($scalar_name, 1, "+$email_flood")
        }
    }
}
```


With a few minor modifications, the code in Example 2 could be updated to assist in security monitoring. If you wanted to check for usernames that are making repeated unsuccessful attempts to login to a machine (brute-force attack), a number of list variables could be created and monitored with thresholds:

- A list using usernames as the key, hit each time a message about an unsuccessful login attempt is seen for that username. Using a LIST ONE threshold would alert when any given username has repeated failed logins (whether on a single host or distributed among many hosts).
- A list using source IPs as the key, hit each time a message about an unsuccessful login attempt is seen for that IP. Using a LIST KEYS threshold would alert when the number of source IPs attempting to connect in a short period of time increases beyond a threshold (e.g. a denial of service attack, or possibly a distributed brute force attack). Using a LIST ONE threshold would alert when a given source IP is making repeated login attempts (e.g. brute force attack).

In **Code Example 3**, the ability to check for the absence of a log message is demonstrated. This functionality is achieved by using an alarm, which is a function provided by the temporal variable server and is implemented using what amounts to an anonymous scalar variable with a threshold attached.

Much like the previous example, the code flows in a way that an average sysadmin with some Perl experience would be able to comprehend. We filter for three types of messages: the job start message, the job finished message, and messages generated by the variable server when thresholds/alarms are generated.

If the message is a job start, the job number is fetched from the message, an alarm is set, and a pair of Perl hashes is used to create a mapping between the job number and the alarm name. If the job finished message arrives before the alarm goes off, the alarm is

cleared and no further action is taken. If the alarm message arrives, we use the mapping to see which job is late, and send an appropriate alert. Further, if the late job actually ends up finishing, we can detect that too by noticing there is no alarm set for that particular job.

In Code Example 3, the alert portion has been shortened into a pseudo-function called `queue_alert()`. In a real VTEC rule engine, the rule engine author would write a function that makes a call to the action server and have it queue an e-mail (or some sort of corrective action).

With a few modifications, this code could have the rule engine comprehend multi-line log messages. For example, ECC errors on Linux systems often appear on multiple lines:

```
kernel: CPU 3: Silent Northbridge MCE
kernel: Northbridge status 940c4002:85080813
kernel:      Error chipkill ecc error
kernel:      ECC error syndrome 8518
kernel:      bus error local node origin, \
      request didn't time out
kernel:      generic read
kernel:      memory access, level generic
kernel:      link number 0
kernel:      err cpu0
kernel:      corrected ecc error
kernel:      previous error lost
kernel:      NB error address \
      0000001a230571d0
```

By utilizing short alarms and a rudimentary state machine in the rule engine, a robust method for capturing multi-line messages like this can be built, which will function even if the log message is incomplete or missing lines. In fact, at AMD we have implemented just such a rule engine that aggregates machine check errors like these and injects new single-line messages into the log stream so yet another rule engine can look for systems with unusually high rates of ECC errors, and close them for repair. In some cases it can even detect which DIMM has failed (by analyzing the syndrome codes) and add that information to the system log in the asset database.

Code Example 3: Checking for Missing Events

```
# syslog-ng metadata
# filter: match("Job.*started") or match("Job.*complete") or \
#         filter(f_thresholds);

my %alarms_by_job;
my %jobs_by_alarm;
while(<STDIN>) {
    my %message = parse($_);
    # If job started, set an alarm so we can alert if it does not finish in 10 minutes
    if($message{message} =~ /Job ([0-9]+) started/) {
        # set_alarm returns the name of the scalar it created
        $alarms_by_job{$1} = $variable_server->set_alarm("+600");
        $jobs_by_alarm{$alarms_by_job{$1}} = $1;
    }
    if($message{message} =~ /Job ([0-9]+) completed/) {
        my $job = $1;
        if($variable_server->get_scalar($alarms_by_job{$job}) {
            # if the alarm is still active, clear it; we're OK
            $variable_server->clear_alarm($alarms_by_job{$job});
            delete $jobs_by_alarm{$alarms_by_job{$job}};
            delete $alarms_by_job{$job};
        }
        else {
            # the alarm isn't active: the job has finished, but finished late.
            delete $jobs_by_alarm{$alarms_by_job{$job}};
            delete $alarms_by_job{$job};
            queue_alert("Job $job finished late!");
        }
    }
}
if($message{message} =~ /TIMEOUT: (\S+) (\S+)/) {
    my ($type, $name) = ($1, $2);
    # One of the jobs didn't finish within 10 minutes, so see which
    # job it was and send an alert.
    queue_alert("Job $jobs_by_alarm{$name} has been running for > 10 minutes!");
}
}
```

Code Example 4 demonstrates the ability to "chain" rule engines. In this example, we have two low-level rule engines, each checking for a different kind of hardware problem on systems. These rule engines use list variables and thresholds to take action when the rate of each type of hardware error exceeds an acceptable threshold. The action taken in these cases is to inject a log message into the log stream that a third, "master" rule engine intercepts. This third rule engine utilizes the key-counting functionality of lists to determine quickly how many kinds of hardware problems a given system has. In the example, a pseudo-action "A" is taken if a system has only one type of hardware problem, while pseudo-action "B" is taken if a system has both.

The net result is that a fairly daunting correlation task is reduced into its three core correlations. These correlations are easily made individually by utilizing temporal

variable constructs in the temporal variable server, making the rule engines easy to write and maintain.

Note the example has been distilled to its core functionality, so is rather simplistic; the same functionality could be achieved in a single rule engine that simply parses more data. However, consider the question of ongoing support and updates. By separating the complex correlation into multiple rule engines, the whole system becomes far easier to maintain over time than with a single, monolithic rule engine. What if you want to start checking for a new class of hardware failure? Rather than modify (and potentially break) a single rule engine, a new (simple) rule engine is created that looks for this new type of failure that injects information about its findings into the log stream. The master rule engine then only needs a minor modification (or, perhaps, none at all) to take advantage of the new data.

Code Example 4: Chaining Rule Engines

```
#### Rule engine 1: Watches for hosts with bad RAM ####
# filter: match("ECC Error") or filter(f_thresholds);
# list threshold will alert when the value of a key exceeds 100.
$variable_server->set_list_threshold("ONE", "ecc_errors", ">=", 100);
while(<STDIN>) {
    my %message = parse($_);
    # an ECC error, hit the appropriate incremter in the ecc_errors list
    if($message{message} =~ /ECC Error/) {
        # We use a 60-second timeout on the hit, which makes the list threshold
        # above alert at an ECC error rate of >= 100 per minute.
        $variable_server->set_list("ecc_errors", $message{from_host}, "+60");
    }
    if($message{message} =~ /THRESHOLD EXCEEDED.*ecc_errors.*\[(\w+)\].*\((\d+)\)/) {
        # We have found a bad host. Generate a log message that Rule engine 3 will
        # pick up that indicates just how bad things are.
        my ($bad_host, $count) = ($1, $2);
        # pseudo-function for brevity; in reality this would queue an inject_msg action
        # that injects a message into the log stream at the given facility and priority.
        queue_alert("daemon", "info", "ALERT: $bad_host with $count ECC errors per minute");
    }
}

#### Rule engine 2: Watches for hosts with bad DISKS ####
# filter: match("EXT3 Error") or filter(f_thresholds);
# list threshold will alert when the value of a key exceeds 20.
$variable_server->set_list_threshold("ONE", "ext3_errors", ">=", 20);
while(<STDIN>) {
    my %message = parse($_);
    # an EXT3 error, hit the appropriate incremter in the ext3_errors list
    if($message{message} =~ /EXT3 Error/) {
        # We use a 3600-second timeout on the hit, which makes the list threshold
        # above alert at an EXT3 error rate of >= 20 per hour.
        $variable_server->set_list("ext3_errors", $message{from_host}, "+3600");
    }
    if($message{message} =~ /THRESHOLD EXCEEDED.*ext3_errors.*\[(\w+)\].*\((\d+)\)/) {
        # We have found a bad host. Generate a log message that Rule engine 3 will
        # pick up that indicates just how bad things are.
        my ($bad_host, $count) = ($1, $2);
        # pseudo-function for brevity; in reality this would queue an inject_msg action
        # that injects a message into the log stream at the given facility and priority.
        queue_alert("daemon", "info", "ALERT: $bad_host with $count EXT3 errors per hour");
    }
}

#### Rule engine 3: Watches for hosts with bad hardware ####
# filter: facility(daemon) and priority(info) and match("ALERT:");
while(<STDIN>) {
    my %message = parse($_);
    if($message{message} =~ /ALERT: (\w+) with (\d+) (EXT3|ECC) errors per (minute|hour)/) {
        my ($bad_host, $count, $type, $base) = ($1, $2, $3, $4);
        # Use a list variable to keep track of the various types of problems a system has.
        $variable_server->set_list("multi_problem_$bad_host", $type, "+3600");

        # If the system has only one thing wrong with it, we take action A, but if there are
        # two things wrong with it we take action B.
        if($variable_server->get_list_keys("multi_problem_$bad_host") >= 2) {
            queue_action("B"); }
        elsif($variable_server->get_list_keys("multi_problem_$bad_host") >= 1) {
            queue_action("A"); }
    }
}
```

This ability to chain rule engines means that extraordinarily complex correlations can be achieved by reducing them to their constituent parts, then chaining rule engines together to track progress through the correlation. Combine this functionality with the fact that rule engines can share temporal variable information through the variable server, and you have an extremely flexible and powerful system for correlating events and taking appropriate actions that is much easier to create, modify, and troubleshoot than other event correlation tools.

5 Useful Rule Engines at AMD

Since VTEC has been in production at AMD for more than four years, we have amassed a significant number of rule engines that perform useful event correlation and self-healing tasks in our computing environment.

Failed Hardware

One of the earliest uses for VTEC at AMD was to look for systems with bad hardware. We found most healthy systems would occasionally report ECC and EXT3 errors, but systems with truly bad hardware would send these errors at a noticeably higher rate. We implemented a rule engine that checks for these messages (the ECC error check uses a chained rule engine that aggregates the multi-line machine check errors that the Linux kernel reports) and then closes bad machines to new compute jobs. Systems in such bad shape that they are streaming error messages into the logs exceed an "emergency" threshold and a signal is sent to immediately power down the affected machine.

NFS File Server Checks

As summarized and simplified in Code Example 2, we have a rule engine that monitors client systems for "NFS server not responding" messages and alerts the storage and networking teams when the rate of messages exceeds a certain threshold.

Reboot Loops

We have all of our servers at AMD configured to send a syslog message when they have finished booting up. A VTEC rule engine watches for these messages and alerts operational staff if a machine is rebooting more than five times in a 90-minute period. These reboot loops can indicate a multi-bit ECC error, kernel panic, or other system configuration problem that requires the attention of a sysadmin.

Interactive Load Monitor Collator

At AMD we have several interactive login servers that are used by our design engineers as gateways into remote datacenters. Some engineers, instead of using our batch scheduling system, will run their jobs directly on the interactive login server, causing performance problems and occasionally even crashing the systems. We have implemented a cronjob that checks for processes that violate our interactive server usage policy, that sends out syslog messages when it detects a process that violates policy. A VTEC rule engine collects these messages, collates them, and generates a daily report for each engineer that is violating the policy with a summary of all of their processes at each site that are in violation of the policy. Additionally, the whole list of users and processes is sent in a daily digest to our interactive server support team, which can determine if it is appropriate to forcibly kill ill-behaved processes.

Out of Memory Tracking

A common problem in AMD's compute grids is out-of-memory conditions. Leaky jobs, or jobs that simply need more memory than is installed on the machine, will cause the system to run out of memory; the kernel then invokes the out-of-memory killer (OOM killer). The syslog message that the OOM killer generates has very little useful data: just a PID and a process name. To identify the owner of the job that caused the OOM condition, we run a cron job every five minutes on our systems that caches the contents of `/proc/<pid>/stat` into a directory under `/var/spool`. When a rule engine sees the OOM killer event, it queues a job ten minutes into the future (to give the server time to recover from the OOM condition) that logs into the machine and fetches the `/var/spool/proc/<pid>/stat` file that was referenced by the OOM killer message. This file tells us not only who was running the errant job, but also how large it was when it was killed. This information is used to craft an e-mail alert to the user asking them to check the status of their batch job submissions to prevent more problems on other compute nodes.

Automatic System Stress Testing

When bringing new systems online in our compute grid, it is important to stress-test them before allowing engineer's jobs on them. We accomplish this at AMD by using a rule engine that watches for syslog messages indicating that a system has completed its automated installation sequence. It then closes the machine to the

batch submission system, kicks off the stress test, and sets an alarm for 12 hours in the future. If the alarm goes off, the stress test must have locked up the system; an operational staff member is alerted to check the system. If the stress test completes (notifying of its status via syslog, of course) the system is either opened for jobs (test was successful) or is left closed (test failed).

Ignoring Flood-Generating Hosts

Sometimes a machine will be so thoroughly broken that it is impossible to shut down the machine, and it just sits there spewing error messages into the syslog stream. This can hamper the VTEC server not only because it can cause it to run out of disk space, but also because it forces syslog-ng to process a flood of irrelevant data. We have implemented a rule engine that watches for hosts sending an excessive volume of log data to the system, and automatically updates the `syslog-ng.conf` file with a filter that drops all of the messages from that host for 24 hours. Once the 24-hour alarm expires, the filter is removed; if the system is still spewing messages, it will be re-ignored within a few minutes, preventing the log flood from adversely affecting performance or data space on the VTEC server.

6 Designing for Performance

The performance characteristics of VTEC can be best described by analyzing the potential bottlenecks of each component, since the slowest single component will likely represent the limiting factor for the overall system. However, due to the multi-core design of VTEC, even if one of the components (usually the variable server) maxes out a CPU, the other components can continue functioning without any degradation on other CPUs.

Syslog-ng

Methods for tuning the performance of syslog-ng are fairly sparsely documented in the reference manual [9]. In the context of VTEC, we are most concerned with the performance of the filtering functions, especially since syslog-ng is a single-threaded process. If syslog-ng spends too much time on the CPU trying to parse

through log filters, it could begin dropping log messages.

We have found after much experimentation that syslog-ng's `match()` filter function is much slower than the other message filtering functions. When rule engines are configured with nothing but a `match()` rule, the syslog-ng process spends all of its time on the CPU, even with a fairly modest (1-2GB/day) rate of traffic. Simply adding one of the "fast" functions in addition to the `match()` function returns performance to acceptable levels, presumably by short-circuiting the `match()` function in many irrelevant cases.

In general, this means that a filter rule like this:

```
match(".*some message.*")
```

can be written better as:

```
facility("daemon") and priority("info")
and match(".*some message.*")
```

to get acceptable performance from syslog-ng.

A typical VTEC server at AMD runs on a four-vCPU virtual machine with 2.9GHz AMD Opteron™ processors under the hypervisor. The syslog-ng instance filters data into 21 rule engines and four local disk logs. The incoming data rate is about 1,000 messages/sec and 10GB/day. With that level of load, the syslog-ng process consumes about 30% of one CPU core, with occasional spikes to above 50% of one CPU core.

Temporal Variable Server

As demonstrated in the examples, the temporal variable server is a critical part of the speed path as log messages route their way through the system. Since virtually every log message that makes its way to a rule engine results in at least one (and many times multiple) requests to the variable server, it is important that it be able to respond quickly.

The variable server is implemented using Perl threads. This allows for simple sharing of the internal database of temporal variables (a hash for

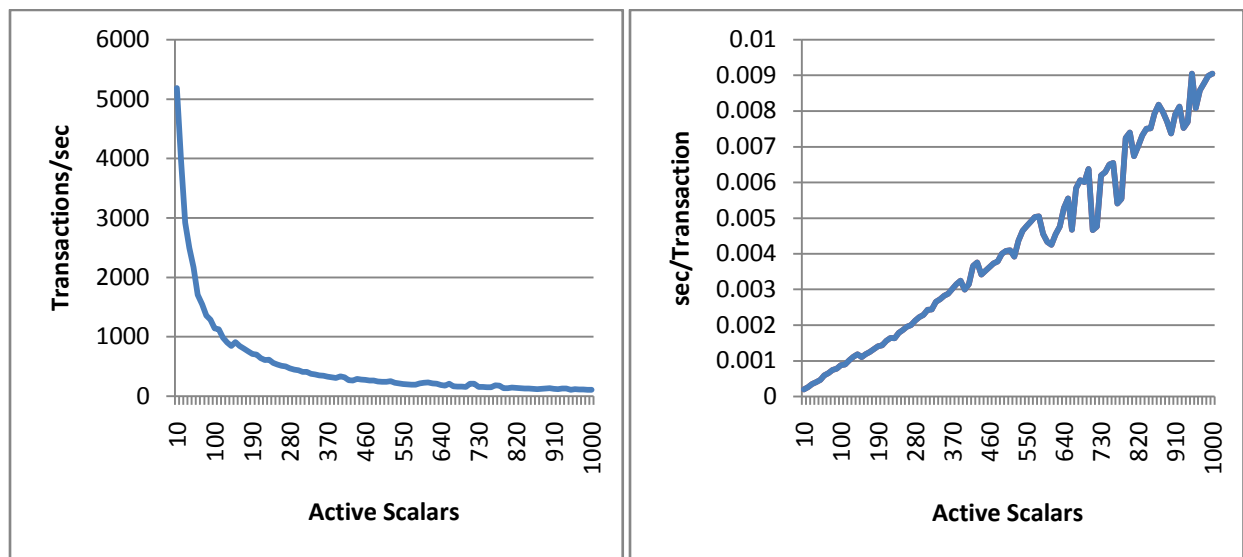


Figure 3: Variable Server Performance Querying a Growing Incrementer

each data type). Six threads run in parallel, using a collection of locks to ensure atomic write access to the internal data structures:

- A server thread that listens for connections (and spawns threads that handle those connections)
- A backup thread that wakes up every ten seconds and backs up the internal state to disk
- Three monitoring threads that wake up every 60 seconds to purge stale data from the internal data structures
- A thresholds thread that wakes up every 60 seconds to scan for variables that have exceeded thresholds.

The persistent threads do not pose any real performance issues; they wake up infrequently, and spend only a few milliseconds to complete their duties before going back to sleep.

The threads that get spawned to service incoming requests are the potential bottleneck. Testing has revealed that for basic requests (e.g. fetch the value of a scalar), performance scales very well. However, as the data structures being queried get more complex (e.g. fetch the value of an incrementer that has 1,000 active scalars in it), performance begins to degrade. Figure 3 shows the transactional performance of the variable server as the size of an incrementer being queried grows in size.

The routine that calculates the value of an incrementer is $O(n)$ with the number of active scalars it contains.

An updated routine that is $O(\log n)$ is being tested and shows promising results, but requires significant code changes and so has not been deployed into production yet.

Rule Engines

Obviously the rule engines, since they can have arbitrary code in them, can be a bottleneck. But the idea behind VTEC is for the rule engines to take advantage of the action server, precisely so they don't have to block for long-running tasks. If the rule engines are coded such that they do not block, they do not represent a bottleneck.

7 Challenges

As flexible and effective as VTEC currently is, there is always room for improvement. We currently face two issues with VTEC.

Variable Server Performance

As described in Section 6, the variable server is the major bottleneck. Since virtually all rule engines depend on incrementers and lists (which are collections of incrementers), the fact that the incrementer data type scales so poorly is a significant hindrance. However, there is new code in the works for the incrementer that should greatly improve its scalability and performance.

Feedback from Actions

The action server is currently a "fire and forget" system. When jobs are queued, the rule engine can get a job ID for tracking, but there is currently nothing useful that

can be done with it. There is no way to query the status of job, or to collect the output of a job, from within a rule engine. Jobs have the option to use `logger` or some other means to inject syslog messages and "phone home" to their parent rule engine, but this process is awkward and is not used in practice.

A major improvement in this area would be to have the action server cache job status information and output for some period of time after the job completes, so rule engines can query for it. Having the action server automatically inject messages into the log stream when jobs complete would also help, because it would give rule engines a trigger to work with to keep up with the jobs they've queued.

8 Conclusion

AMD has used VTEC since 2006 to monitor and automate maintenance activities on its large compute grids. Log volumes range up to 10 GB/day with VTEC running smoothly on modest two- to four-core virtual and physical machines. VTEC tracks hardware problems such as disk, CPU, and RAM failures and takes appropriate actions (e.g., shut down/close the broken system and create a ticket). VTEC can monitor the environment for trends that indicate events (e.g., n systems are unable to contact m NFS filers, so there must be a network problem). Most importantly, VTEC enables autonomic computing by allowing intelligent dispatch of repair jobs in response to detected problems. If these repair jobs fail to work, VTEC can notify humans to take over.

In summary, VTEC is a powerful tool for automating log analysis and event correlation. While there are many other tools that perform similar tasks, VTEC's approach to the problem presents a complete, scalable, and intuitive solution that is able to grow and adjust to virtually any workload.

9 Author Biography

Paul Krizak is currently a member of the technical staff at AMD and has been a professional systems engineer for more than seven years. His duties at AMD have generally surrounded infrastructure support of the large engineering research and development grids, most commonly in support of the autonomic Linux OS provisioning and configuration infrastructure. He graduated with a bachelor's degree in computer science from Texas A&M University in 2005.

10 References

1. Rouillard, John P. "Real-time Log File Analysis Using the Simple Event Correlator (SEC)." *Proceedings of LISA XVIII* (2004): 133-49. Print.
2. *SEC - Open Source and Platform Independent Event Correlation Tool*. Web. 22 July 2010. <<http://simple-evcorr.sourceforge.net/>>.
3. *Splunk | IT Search for Log Management, Operations, Security and Compliance*. Web. 22 July 2010. <<http://www.splunk.com>>.
4. M. Burgess. On the theory of system administration. *Science of Computer Programming*, 49:1, 2003. Print.
5. *Cfengine - Automatic Server Lifecycle Management*. Web. 22 July 2010. <<http://www.cfengine.com>>.
6. "Syslog Server | Syslog-ng." *Gateway Solution | Network Security | BalaBit IT Security*. Web. 22 July 2010. <<http://www.balabit.com/network-security/syslog-ng/>>.
7. Hansen, Stephen E., and Todd Atkins. "Automated System Monitoring and Notification with Swatch." *Proceedings of LISA VII* (1993): 145-52. Print.
8. "Logwatch on SourceForge.net." *SourceForge.net*. Web. 22 July 2010. <<http://sourceforge.net/projects/logwatch/files/>>.
9. "Syslog-ng V2.0 Reference Manual." *Syslog: Main/Home Page*. Web. 23 July 2010. <<http://www.syslog.org/syslog-ng/v2/>>.
10. Finke, Jon, "Process Monitor: Detecting Events That Didn't Happen," *Proceedings of LISA XVI* (2002): 145-153. Print.