RedAlert: A Scalable System for Application Monitoring

Eric Sorenson – Explosive Networking
Strata Rose Chalup – VirtualNet

ABSTRACT

RedAlert is a complete application monitoring system which consists of a stateful server daemon and extensible Perl client API. Almost any IP-protocol service is a candidate for RedAlert monitoring: the clients determine what error condition they have discovered, convert that information into a standard message format, and transmit the Alert to the server.

RedAlert therefore will easily plug in to existing script-based monitoring environments, providing greatly increased functionality for a minimal investment in configuration time. This functionality includes volume tracking, interval sampling, threshold-based notifications, and reporting mechanisms which include pager, electronic mail, and SNMP traps.

We have chosen to focus on email monitoring, specifically postmaster bounce mail, for the scope of this paper. Bounce mail is both ubiquitous and complicated, making it ideal for RedAlert monitoring.

View from 25,000 Feet

RedAlert is an extensible, easy-to-use client/server framework written in object-oriented Perl. It comes with a couple of sample client programs and classes, a generalized API to create new kinds of clients, and a full-featured server which supports several types of threshold monitoring and notification channels.

The goals in its design were:

- easy integration into existing monitoring
- extendability on the client side
- the ability to catch failure modes which slip by traditional network monitoring systems undetected

True to its client/server nature, RedAlert is composed of two parts: the monitoring daemon and a set of clients which report to it. This enables a RedAlert installation to aggregate information about specific types of occurrences and trigger events based on administrator-configured thresholds.

Some of the highlights of the implementation:

1. Flexible connection model. RedAlert uses Perl's Data::Dumper [1] and TCP connections to pass objects across the network. The connection model doesn't care about an Alert's content, just its "well-formedness," which promotes extensibility.
2. Object oriented. RedAlert was built from the ground up using OO methodologies. If you have home-grown Perl scripts for automation, you'll find it easy to extend RedAlert's monitoring capabilities to your site. For example, given a Perl program which connects to a web server and checks for a known-good Content-length: header, it's ten additional lines of code to send a RedAlert warnings if any of the stages of the connection process fail.
3. SNMP notifications. RedAlert has several methods of triggering SNMP traps. Browsing support is coming, but is not implemented at the time of this writing. RedAlert has been allocated a registered branch of the 3Com Enterprise MIB. You are in no way obligated to download the 3Com MIB or use 3Com equipment. The important part is that notifications are tagged with unique registered ObjectID's so you won't have any conflicts with your existing SNMP setup.

Origins of RedAlert

Our system originally came about as a response to a site-specific need. The authors were coworkers at the site of a client who was building an Internet Access Service using some Netscape products, including Netscape Messaging Server. A few months into the beta test phase, we realized we were being overwhelmed by the amount of bounce mail the NMS' would generate. Load tests brought this problem into sharp relief: after 13,000 "user over quota" bounces from a test gone haywire brought down the operations mail server, we decided it had gone far enough.

We sensed the need for an intelligent front end to the postmaster mailbox. This front end would keep track of the types and quantities of bounces it received and determine whether they represented a "blip," a trend indicating a more serious problem, or an error which required immediate sysadmin attention. One "user not found" error probably isn't serious; thirty of them to the same user in the span of a few minutes, on the other hand, might indicate a mailbomb in progress. The client/server model lent itself to a polymorphic network of application monitoring programs, with the client scripts communicating their results to a server which could make decisions on what a given error means.

The Status Mail Deluge

Many service daemons are equipped to report errors or unusual conditions via email. In the halcyon days where every user was also her or his own sysadmin, this was a friendly and useful way to report errors. The information would appear in your mailbox, perhaps even *biff*(1)'d across your screen.

With a typical crop of cron jobs, license monitors, network service daemons, and user email, even a small cluster of several to a dozen machines can generate a healthy quantity of email over the course of several days or a weekend. A mid-size engineering firm of two to three thousand employees may have several hundred servers and generate as much email in a day as our small cluster would in a month.

User service clusters, such as those deployed by Internet Service Providers, typically run a number of email-noisy security tools in addition to the normal utilities. Their volume, however, is dwarfed by that of postmaster mail. Running electronic mail services on an ISP scale can literally result in megabytes of postmaster mail per day.

From "Needs Practice" to "Best Practices"

In many small shops there is something of a laissez faire attitude towards email status notifications: "If something goes wrong enough, the users will tell us."[1] At the other end of the spectrum are the shops which have everything sorted, scripted, and routed. In ascending order of utility, here are the methods currently employed by most sites:

1. Check each server individually "when you get around to it." This is usually shorthand for "we will check postmaster/root mail when something breaks."
2. Define aliases on each host to funnel mail to a central collection point, i.e., alias root@thishost to root@mailhub. Read by hand in between fire-fighting.
3. Implement the centralized funneling, and add some form of regular expression filtering to sort mail into different files/folders. Procmail is a typical method, followed closely by the filtering options native to "whatever mailer the lead sysadmin prefers."
   At some sites, the processing is delayed rather than real-time, and the filtering is invoked regularly on the common inbox to perform the sorting. Read between fire-fighting, or have a junior team member keep an eye on the folders every day or two.
4. Implement item 3, and add scripting. The scripting seems to take one of two forms, and it is unusual (though praiseworthy) to see both employed at the same site.
   - Add an extra step or two to the filtering, sending particular messages to a team member's inbox or to the email input of a trouble-ticket system such as Remedy, RT, req or Scopus.
   - Add one or more cron jobs that rotate the types of folders (e.g., host-not-responding, user-not-found, etc) daily and watch the size of the folders between rotation. If a particular file gets larger than a site-specific "typical" size, notify via email to a pager gateway or trouble ticket system.

In practice, this means that site policy will usually be extreme in one direction or the other. Some sites will save everything, but practice "file and forget" or "dig through the attic in case of trouble." Other sites will turn off postmaster copies of bounces, set logging options down to "critical only" for servers, and flood some unlucky soul's pager or regular email box with everything that comes through.

Putting the "State" back in "State of the Art"

The critical piece which all of the above lack is an "intelligent" collection node. To qualify, a node should receive the incoming messages and be able to make connections between them based on content: their origin, destination, meaning, overall number, frequency during an interval, and so on. A solution relying on scattered individual files and filter-triggered scripts doesn't count.

All these sets of stateless, event-driven scripts are still the equivalent of counting elephants in a field with a fiberglass pole. You walk with the pole held out beside you, and every now and then it deflects on something, bends back, and smacks you in the face. You then page your postmaster with an "elephant spotted in Field Seven" message. It's up to the postmaster to keep track of how many pages came in that day, and which fields are rife with elephants.

Enter RedAlert's stateful server daemon: keeping tabs on troublesome elephants day and night. The client API allows for extensibility and greater sophistication in types of clients monitored, and is (in our humble opinions) a very useful contribution. Fundamentally, though, the server daemon serves as the keystone to the arch of intelligent status mail processing, transforming it from a pile of semi-organized building blocks into a definite structure.

For the rest of this paper, we'll walk through building a RedAlert client and server configuration to monitor a typical application, namely, bounce messages generated by the Sendmail daemon.

[1] This meshes particularly badly with what Elizabeth Zwicky calls "systems administration via psychic powers" but can be a useful predictor of an individual shop's normal uptime/downtime ratio.

RedAlert Systems Architecture

Since RedAlert can be deployed in a centralized or distributed fashion, you must decide how you wish to aggregate the alerts. This requires identification of your goals for the monitoring of your mail system.

Obviously if your site is a high-volume site you will have different goals than if you are running a small site. Individual local hosts or particular remote destinations may be of specific importance to you, and you may wish to have someone paged if ''host not found'' bounces begin popping up for that site. On the other hand, you may simply wish to have statistics recorded on the ''miss'' rate of user addressed email.

If you have deployed load-balanced banks of mail servers, you will probably wish to have a separate RedAlert server instance aggregating traffic for each bank. The prioritization of alerts for the banks may depend largely on upon whether your load-balancing solution can ''busy out'' an unresponsive server or if it must be done by hand.

Decide where to aggregate

You can either aggregate the information yourself by redirecting postmaster mail to your mailhub, or you can make the service machines do the work and use an SNMP collection tool for historical data and trend analysis. We recommend the latter in most cases, as it localizes and distributes the overhead of running the RedAlert service.

The approach you choose will be determined by your desired handling of the actual postmaster mail. If you are operating under a ''notice, then delete'' policy, it is of course more efficient to throw away the mail right on the originating server. Errors of a type which should be saved can presumably be redirected to a central server by the same filtering approach which you are using to invoke the RedAlert client.

Option 1) Centralize all mail to a mailhub, run it on the inflows, snmp monitor mailhub.

Option 2) Run it locally on the servers, (optionally routing a copy of the message to a central collection point for archiving), snmp monitor each server (which you're probably doing anyway).

Option 3) Run a distributed setup, with a config master for each class of service machine, aggregating via option 1 or option 2, or an option 3 recursive of ''mailhub for class.''

Decide What To Throw Away

Types of postmaster mail which a large site might wish to track but throw away include ''hostname not found'' and ''user not found'' originating at remote sites. It can be beneficial to know the numbers and frequency of these sorts of errors, but there is rarely information in them useful to maintaining your mail servers.

For example, an unusually large spike in ''hostname not found'' is worthy of an alert, as it could signal the demise of a DNS server on which that mail machine depends, the useful information is really in the fact that the spike occurred, not the content of the individual messages.

Typical Postmaster Alerts

Many of these are in the class of ''Gee, if anyone actually read postmaster mail at our site, we would have seen this when it came through.'' This is exactly why we came up with the idea of RedAlert – because so few sites actually examine postmaster mail in a timely fashion.

We will use sendmail as an example, given its widespread acceptance across many types of installations. In our notification examples, it is arguable that any condition severe enough to generate an SNMP trap might also be a condition where you would want to page someone. However we will assume that the SNMP monitoring system has its own rules for paging, and that some trap-only notifications might result in pages. There are some things that are sufficiently bad that we would want to page anyway and risk double-paging some poor sod at 3am.

Please note that our examples do not constitute a comprehensive list of any and all errors, nor are the examples ranked in any particular order, most especially including likelihood or severity.

Severe MTA Error

Most types of 451 errors should cause system staff to be paged. Of course, some of them may indicate an OS-level problem of a severity that would preclude the operation of RedAlert as well. A prime example is ''451 %s: cannot fork'', which we'll use to build up our RedAlert configurations. See Table 1.

Performance Related Conditions

These are things which are useful to log or trap and later correlate against system and network load

Severe MTA Errors		
Error	Threshold	Notification
451 %s: cannot fork	0	Trap, Page, Log
452 Error writing control file %s	N per interval	Trap, Page, Log
452 Out of disk space for temp file	N per interval	Trap, Page, Log
451 %s: lost child	N per interval	Trap, Log

Table 1: Severe MTA errors.

data. This will let you see some cause and effect linkages more clearly than random poking. Many of these errors will never be seen at small or well-scaled sites. Tracking the remote host involved in timeouts will allow you to set your mailer timeouts to handle certain destinations which can be notoriously slow. A very high-volume site may wish to use mailertables to segregate this traffic to dedicated mail routers with unreasonably high timeout values. See Table 2.

Potentially Security Related Errors

Some mailer errors are most often seen in response to root-kit style cracking attempts. These in particular often involve strange terminations in the mail.local phase. Generating a trap for these errors gives NOC staff the ability to investigate in real-time.

Others of this category of error can be generated in response to spammers attempting to use the MTA for spam dumping. Logging "452 Too many recipients" and correlating against RADIUS authentication logs may enable ISP abuse desk staff to identify spammers. Some of the spam dumping programs out there are also poorly written and generate bad SMTP.

Some of the threshold values in our table might be better off with "N per interval" or even "Y rate of increase" instead of using single-Alert triggers. and which should just be "after some small number." This will depend largely on just how huge your site is and how peculiar your mail clients may be. Sites running cc:mail or Exchange gateways for users who love forwarding and love attachments may see these kinds kinds of errors on a fairly routine basis, for example.

Deploy Clients

Create clients for the various individual scripts or add cases in your existing filtering script for routing postmaster (root, daemon, cron, etc) mail. Substitute or add a RedAlert client script with the appropriate arguments and push the configuration files and modified filtering scripts out to the clients.

Performance Related Conditions		
Error	Threshold	Notification
451 open timeout on %s	N per interval	Trap, Log
451 reply: read error from %s	N per interval	Trap, Log
451 timeout waiting for input during message collect	N per interval	Trap, Log
452 Insufficient disk space; try again later	N per interval	Trap, Log

Table 2: Performance related conditions.

Potentially Security Related		
Error	Threshold	Notification
452 Too many recipients	0	Trap, Log
500 Bad usage	0	Trap, Log
051 WARNING: writable directory %s	0	Trap, Log
451 %s: died on signal %d	0	Trap, Log
550 Access denied		
550 Address %s is unsafe for mailing to programs	0	Trap, Log
550 User %s@%s doesn't have a valid shell for mailing to files	0	Trap, Log
500 Parameter required	0	Trap, Log
500 smtp: unknown code %d	0	Trap, Log
501 Syntax error in parameters scanning "%s"	0	Trap, Log
501 %s parameter unrecognized	0	Trap, Log
501 %s requires domain address	0	Trap, Log
501 Unknown BODY type %s	0	Trap, Log
502 Sorry, we do not allow this operation	0	Trap, Log
503 Nested MAIL command: MAIL %s	0	Trap, Log
503 Polite people say HELO first	0	Trap, Log

Table 3: Potential security errors.

While the client information could all be included in one master file, we recommend splitting it into service-specific and/or class-specific config files. Thus a mistake in editing your squid_proxy.conf file would not affect your smtp-in.conf, etc. This approach arguably makes deployment easier, as a large site could build up a library of different client modules to reuse and recombine for new situations.

The Configuration Files

Now that we know what we're looking for, it's time to start figuring out how to find it. "Finding it" in RedAlert's case means setting up your client and server configuration files in a consistent, complete, and logical manner.

RedAlert uses simple, text-based configuration files for both clients and the server. As mentioned above, it's generally better to keep a client's configuration pared down to only those Categories of alert which that client will be expected to handle; you win both in ease of use and execution speed – less to parse means less memory and time used in parsing.

Our configuration files look similar to a Windows .INI file. They contain section names in square brackets that identify the Category we're handling, and then list "name = value" pairs for relevant variables for that section. Using AppConfig, a Perl module by Andy Wardley [3], allows us a great deal of flexibility with config file parsing, so the niggling typos which plague primitive parsers aren't a problem.

The configuration file is divided into three main sections.

1. Global. The global section contains overall config information such as port number, system-wide defaults, etc.
2. Panic. The "panic" section contains configuration information for RedAlert to report errors on itself. Problems parsing an incoming client messages, failure to contact the server's RedAlert port, or even debugging strings can be sent out with a Panic, typically an email message sent to a host of last resort.
3. Category-specific. The rest of the file's sections contain the site-configurable settings for types of events, notification procedures, etc. These sections on the client side only contain a template for the Alert to send; on the server, they set up monitoring threshold and notification parameters.

Please note: It's very important to keep your Categories consistent between the clients' configuration files and the server's. An incoming Alert's Category determines which threshold will be incremented and processed. RedAlert tries to do something reasonable with unfamiliar Categories, but "reasonable" means sending a Panic message saying it didn't know what to do with the Alert. Receiving a large number of Panic messages due to a misconfigured client will dilute the legitimate Panics . . . in a truly degenerate situation, you might even be forced to run a meta-RedAlert server to filter incoming Panic messages!

We'll build up sample client and server configurations for monitoring our sendmail alerts as we discuss each component in turn.

Crafting a RedAlert Client

The first (and trickiest) part of making a RedAlert client is figuring out exactly what it is you're looking for. In the case of our sample client for parsing Sendmail bounces (hereafter called sendmail_client.pl), we poked through the sendmail binary with *strings*(1) and came up with appropriate regular expressions to match the various kinds of bounces the program can generate.

We then created a simple sendmail_client.conf, with templates for each of the different Categories of bounce we might receive. In the interest of brevity, this client example will only focus on one specific error from the tables above: "Cannot fork". This is a pretty serious error condition, indicating that sendmail had used up its allotment of child process id's, or (worse) that the machine's process table was full. We'll want to extract the "%s" substitution from the message and use it to fill in our Template; since we're dealing with sendmail here, the Message-Id of the bounce will come in handy too. So, our sendmail_client.conf looks like the code in Listing 1a.

The methodology you use to create and configure your own clients will depend upon the kinds of error strings your application produces – in the case of a Squid web-proxy monitor, as another example, the error conditions are indicated by known-bad responses

```
[global]
port = 7200
debug = 1
servername = alerthost.ops.domain.com
lastresort = redalert-admin@ops.domain.com

## Types of email bounces
[sendmail_cannot-fork]
template = "Sendmail couldn't fork doing _AAA! Message-ID: _BBB"
```

Listing 1a: sendmail_client.conf.

from the server (a 404 or 500 error to a page which ought to be there or a CGI which should be working), or by a failure in the various stages of establishing the TCP connection to the proxy's port. This level of monitoring is what makes RedAlert special – no matter how expensive the network monitoring solution in use at sites we've seen, it always seems to miss certain failure modes that always end up biting you in an uncomfortable place.[2] With RedAlert, however, you can catch errors specific to your services, plus adapt your monitoring routines to the new and interesting failures that seem to creep in with a new code revision or configuration change.

To promote reusability, we created a new subclass of RedAlert::Client called Sendmail. This translates into a file under the RedAlert library directory: RedAlert/Client/Sendmail.pm. To avoid losing sight of the more typical cases where you're integrating RedAlert client functions into already-existing programs or writing a simpler frontend that won't involve package creation, we'll focus more on the sendmail_client.pl interface and the Client API, delving into the new subclass' methods only when necessary. Consider the following code segments from sendmail_client.pl shown in Listing 1b.

The first two lines pre-load our module, then instantiate a new RedAlert::Client::Sendmail object. The "new" constructor method inherits attributes from both its parent classes, the base RedAlert class and the Client subclass; see Listing 1c.

This line calls the Configure method to parse our config file, loading up the categories and templates into which this bounce might fall. Configure is actually a RedAlert::Client method which we have inherited; see Listing 1d.

This is our only chunk of real client-specific code. Since we're destined to be passed an email, we can use Mail::Internet's builtin facility to create a new Mail object by reading from stdin, the odd-shaped [<>] construct in like 10. We use some handy methods in the Mail module to create references to Mail::Header and Mail::Body objects from the incoming email in lines 6 and 7, and pass these as arguments to the Parse method of our Alert object.

Here we'll have to descend into Sendmail.pm for a bit, to follow the Parse method as it fills in our Template with appropriate substitutions for the placeholders defined there. See Listing 2a for some code fragments from Sendmail.pm.

Line 2 might seem puzzling at first, but it starts to make sense with the knowledge that calling a method with the arrow operator, as in "$alert->Parse", passes the object's reference as the parameter to the method. So when we "shift" above, we're assigning the the first element of the implied @_ to $self. This gives us a local copy of the object to manipulate. The next two elements of @_ are references to the Mail::Headers and Mail::Body objects passed to us by the sendmail_client.pl; see Listing 2b.

Here's the meat of the bounce processing. We loop through each line in the body of the message, scanning for error codes and their attendant strings indicating specificity. The conditional shown in line 5 will match our 451 errors, using the (\w+) pattern to match the string from sendmail indicating what it was trying to do when the fork failed. Once we know what Category this bounce belongs to, we send it to SetType (line 7), which updates the Alert object's state and pre-loads the appropriate Template. Since the message-id of sendmail bounces helpfully include the hostname and timestamp, it's all we need to uniquely identify this Alert. We extract it from the Mail::Header object in line 8, and then pass both of these variables into the Substitution method, which iterates through its list of parameters and assigns them to the _AAA .. _ZZZ patterns in this category's Template, in order. Remember, our template for "sendmail_cannot-fork" Alerts looks like Listing 2c.

There's one conditional for each possible Category which we know about, and a final catch-all line

[2]Like the back of a Volkswagon. Never mind.

```
1 use RedAlert::Client::Sendmail;
2 $alert = new RedAlert::Client::Sendmail;
```

Listing 1b: Code segments from sendmail_client.pl.

```
3 $alert->Configure("/opt/src/redalert/nms_client.conf");
```

Listing 1c: Code segments from sendmail_client.pl.

```
4 use Mail::Header;
5 my $mail = new Mail::Internet( [<>] );
6 my $headers = $mail->head();
7 my $body = $mail->body();
8 $alert->Parse($headers, $body);
```

Listing 1d: Code segments from sendmail_client.pl.

in case we're at the end of our rope and still don't have a match: see Listing 2d.

Back in sendmail_client.pl, we have only to send our completed alert off and we're through.

```
$alert->Send;
```

Send creates an eval()-able chunk of perl from the useful parts of our object, namely the Category and Specificity, and fires it off to the server . . . and that's it! To recap, a minimal RedAlert client, which sends an alert containing the date every time it's invoked, might look like this code from minimal_client.pl:

```
use RedAlert::Client;
$alert = new Client;
$alert->Configure(
      "/path/to/my/client.conf");
$alert->SetType("irritating");
$alert->Substitution( ‘date‘ );
$alert->Send;
```

This assumes that client.conf looks something like this:

```
[irritating]
template = "minimal_client".
        " was invoked at _AAA"
```

The RedAlert Server, Close Up and Personal

The RedAlert server is the guts of the system. It's the destination for all your different clients' Alerts, and as such needs to know how to track the various categories of alert and when and how to send a notification if a threshold is knocked over. The server (redalert.pl) normally runs as a daemon, but can be put into non-forking mode to facilitate debugging.

Server Configuration File

The RedAlert server config looks syntactically similar to the client configuration file, but contains a lot more information. The section headers are the names of **all** categories, system-wide, for which we might receive an Alert, plus sections for global variables and Panic configs). Unlike the client configuration file, we aren't interested in Templates to fill out for the various Categories; rather, we need to determine what to do when an alert of a particular Category is received. As such, while a client's config may contain only Categories which that particular client will be handling, the server config should contain a section for every category of Alert, system-wide. Again, unconfigured categories indicate that one of your clients is sending mismarked Alerts and will result in a Panic notification. This ‘‘last resort'' notification means we don't drop anything on the floor, making it easy to track down those elusive ‘‘telnet'' problems.

All server config files should have a [global] section, to define the port and interface/hostname to bind, an Accounting Interval to summarize and mail statistics reports, and a [panic] section for self-reporting and as an address of last resort for unconfigured categories problems delivering Notifications. Listing 3 shows the start of a typical config (redalert.conf).

```
sub Parse {
    my $self = shift;
    my ($headerref, $bodyref) = @_;
```

Listing 2a: Code fragments from Sendmail.pm.

```
    foreach $line (@$bodyref) {
         if ($line =~ /451: (\w+): cannot fork/) {
             chop $line;
             $self->SetType("sendmail_cannot-fork");
             my $messageid = $headerref->get("Message-ID");
             $self->Substitution("$1", "$messageid");
             return $self;
         }
         elsif { [ ... ]
    }
```

Listing 2b: Code fragments from Sendmail.pm.

```
template = "Cannot fork on operation _AAA! Message-ID: _BBB"
```

Listing 2c: Template for sendmail_cannot-fork.

```
    $self->Panic("Got an unparsable message: @$bodyref");
    return $self;
}
```

Listing 2d: Final catch-all line.

```
[global]
debug               = 1
acct_interval       = 86400s
port                = 7200
servername          =
          alerthost.ops.domain.com
threshold_interval  = 60
alert_host          =
           mailhost.ops.domain.com
[panic]
alert_type          = email
alert_dest          =
     redalert-admin@ops.domain.com
[sendmail_cannot-fork]
threshold_trigger   = 1
threshold_interval  = 0
alert_dest          =
         duty-pager@ops.domain.com
[sendmail_complicated-threshold]
alert_type          = email
alert_dest          =
           redalert@ops.domain.com
threshold_interval  = 5m
threshold_trigger   = 20
threshold_delta_max = 5
threshold_delta_min = 3
threshold_average   = 4
send_only           = 3
send_after          = 2
```

We'll explain what each of the category-specific configuration lines means in a bit; suffice it to say for now that each "451: Cannot Fork" Alert will result in an email page to "duty-pager@ops.domain.com" immediately upon receipt.

Event Loop

redalert.pl uses Joshua Pritikin's excellent Event [4] package to neatly solve some of the thornier problems in running long-term servers in Perl. Event provides "a central facility to watch for various types of events and invoke a callback when these events occur" [from the Event .pod]. A watchable "event" can be a timer-based interval hitting zero, activity on a file or network socket, a signal sent to the process, a variable incrementing or changing its value, to name a few. A callback is simply a subroutine which acts on the information received by its watcher.

The important thing about Event is that its watchers are processed asynchronously, queued for execution and then run sequentially based on their

```
use RedAlert::Server;         # Server methods
use RedAlert::Threshold;      # All the Threshold methods

use Event qw(loop unloop);    # import loop(), unloop() into namespace
require Event::tcpserv;       # JPRIT's server shortcuts
require Event::timer;         # Watcher type for countdowns
require Event::signal;        # Classy signal handlers

$server = new Server;         # instantiate myself

$configfile = "/opt/src/redalert/rasv.conf";

# Sanity checking on the values, add a reference to the AppConfig object
$server->Configure("$configfile");

# Create state for each category we'll be monitoring, add a reference
# to the Threshold object which contains them.
$server->InitThresholds;

# Daemonize forks us, global_debug from the configfile via AppConfig
if ( $server->Daemonize || $server->Config("global_debug") == 1) {
    my $reread_config = new Event::signal( 'e_signal', 'HUP',
                          'e_cb', \&ReParse,           # the callback
                          'e_desc', 'reread_config' );  # description
   # Server method to update ourselves
   $server->AddWatcher('reread_config', $reread_config);
   loop() || $server->Panic("Couldn't start the server's main loop!");
}
```

Listing 3: Typical beginning of server, redalert.pl.

priority; this defers the problem of race conditions and is a lot more portable than using threads (which Perl can't do very well anyway).

RedAlert Startup and Watchers

To setup the Event loop, we first import the modules we'll need, then register our starting Watchers (the events we'll be acting upon), and finally start the loop itself. The start of the code for the server is show, in part, in Listing 3 (with commentary inline).

While there are subroutines below for handling different Watchers' callbacks, this section is really the heart of the server program. There's a lot going on here, but it demonstrates a clean object-oriented API. The containment of two other types of objects (RedAlert::Threshold and AppConfig::State) within the RedAlert::Server object lets us encapsulate their internal structure, in accordance with the OO principle of "information hiding." Our Server methods provide a higher-order wrapper to the objects' data, so their structure can change internally (say, to store the Thresholds in a database instead of in memory), but as long as the API remains consistent, the program itself won't notice the difference.

Thresholds

Let's concentrate for the moment on the variables beginning with "threshold_" in the sample config file above. Note that at no time do you specify in the the config what kind of threshold you want; you simply define whatever threshold variables you're interested in watching, and RedAlert does all the work. Let's walk through each of the threshold types in turn, starting with the simplest kind. All threshold variables in the configuration file start with "threshold_"; we'll give a visual representation of each type and then a sample config which would create that kind of threshold.

Summary: The simplest kind of threshold is no threshold at all – if you define an interval but no trigger, RedAlert will simply keep statistics on that Category, sending you a summary of the number of Alerts received over the interval. Summary counters are cleared at the end of the interval. See Figure 1.

In a config file, this is as simple as setting the threshold_trigger to zero, i.e., "don't ever trigger a Notification." If your fall-through threshold_interval in the [global] section is appropriate, zeroing out the trigger is the only thing you need to define. The following lines would create a Summary threshold:

```
[category_type]
threshold_trigger  = 0
```

Trigger: The inverse of the Summary threshold is Trigger, which sends a notification upon each Alert the server receives. This is useful for really desperately bad kinds of problems, which you don't want to track for statistical purposes, you just want to know about it right away. See Figure 2.

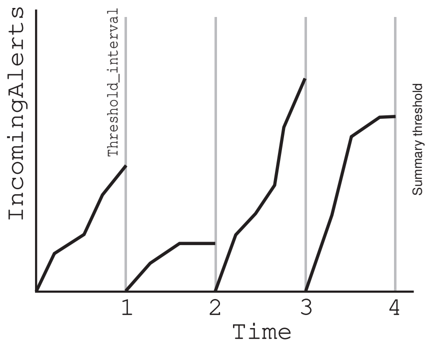

Figure 1: Summary threshold.

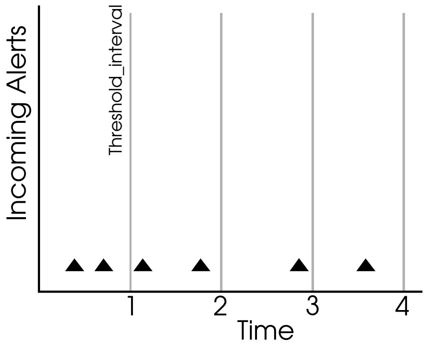

Figure 2: Trigger threshold.

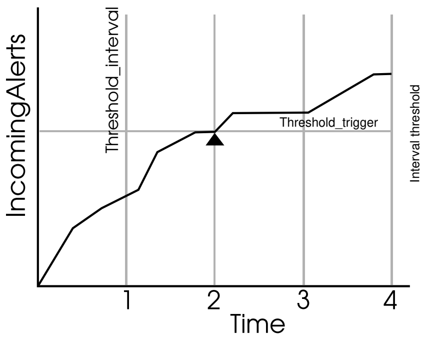

Figure 3: Interval threshold.

To set up this type of threshold, simply set threshold_trigger to be 1.

```
[category_type]
threshold_trigger = 1
```

Interval: An Interval threshold tracks the frequency of incoming alerts and sends a notification if the server receives more than x notifications in y time. A possible use for Interval thresholds is monitoring response times from applications where it's OK for them to sometimes bog down a little bit, but when you receive, say, ten Alerts in ten minutes saying the response time was over five seconds, you'd want to know about it. We'll use these parameters for the graph in Figure 3.

As the graph implies, we'd define both threshold_trigger and threshold_interval for this kind of Alert to be ten, like so:

```
[category_type]
threshold_trigger = 10
threshold_interval = 10
```

Maximum Delta: The greek letter delta signifies change, and with this type of threshold we're monitoring the maximum change in time between receiving two Alerts of the same Category. This is useful for uptime or heartbeat monitoring; you'd run your client as a cron job, say every five minutes, and it would send an "all clear" message once it finished processing. If more than six minutes pass between one notification and the next, we'd want to know about it, which is where Maximum Delta comes in. See Figure 4.

```
[category_type]
threshold_delta_max = 5m
```

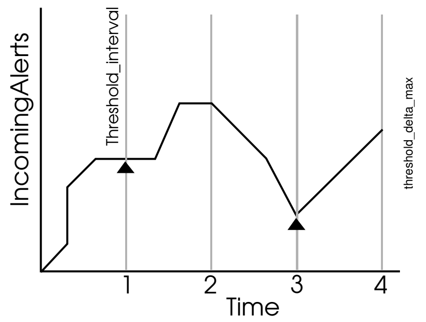

Figure 4: Maximum delta threshold.

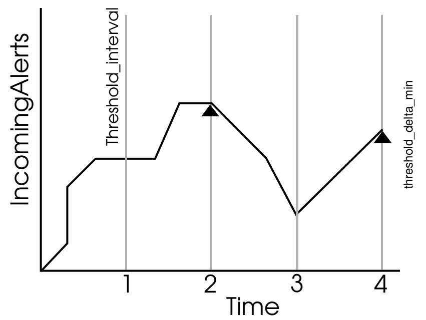

Figure 5: Minimum Delta Threshold

Minimum Delta: Like the Max Delta above, Minimum Delta measures the time between received Alerts of a given Category. However, in this case we want to receive a notification if the Alerts start coming in too quickly instead of too slowly. A potential use for this type of threshold might be monitoring "user not found" errors from a mail server, where a few of them spaced out over a whole day is fairly typical, but ten in rapid succession might indicate a misdirected mailbomb or denial-of-service attack. See Figure 5

```
[category_type]
threshold_delta_min = 3m
```

Average: The most complicated (arithmetically, not configuration-wise!) type of RedAlert threshold keeps a running average of the rate the server is receiving alerts of a particular type, and sends a notification if the average exceeds the configured value. This is quite similar to the idea of "load factor" on UNIX machines, and can be used in the same situations where you'd normally monitor the load average. Average thresholds are more complicated than Interval thresholds because they don't use an absolute value for the trigger; rather, they keep *relative* values and thus provide greater flexibility for monitoring things like proxy transactions per minute.

$$\text{current\_average} = \frac{(\,(\text{interval} - \text{current\_delta}) * \text{old\_average} + 1\,)}{\text{interval}}$$

The threshold_average config variable is used. It's a numeric value which expresses the number of Alerts received over an interval value. As such, you need to define the threshold_interval as well as threshold_average to configure this type of alert.

```
[category_type]
threshold_interval = 60s
threshold_average = .5
```

Note that thresholds are additive wherever possible: except for ones which trigger a notification on each Alert received or only do statistics-gathering, RedAlert will check the most complex type of threshold first, and then continue down the list to make sure none of the lesser-order thresholds are triggered too. This means that if you define a threshold_trigger and threshold_interval as well as threshold_delta_max for a particular Category, this will create a Maximum Delta threshold plus an Interval threshold for that type of Alert. This allows you to simply and quickly set up a very powerful monitoring profile for your server.

Notifications

So, once a threshold is triggered, what next? There are two kinda of Notifications which can result from a new Alert the server receives: email and SNMP. Each of these methods will prepend its own message onto the text of the Notification, so the worst-case Notification you receive will say something like Listing 4.

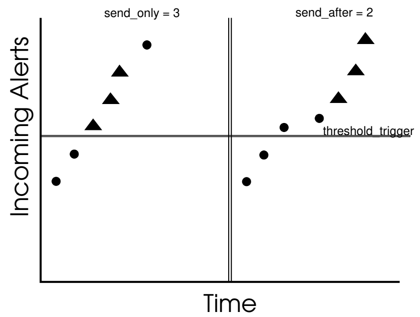

Figure 7: Send After and Send Only

Line 1 is from the Panic method, Line 2 from Server::CheckThresholds, and line three was the original Alert we received.

All categories can specify two attributes which will reduce likelihood of being ‘‘spammed’’ with RedAlert notifications. send_only will increment a counter upon each trap sent, and only send up to this many notifications in one accounting interval. send_after will wait until receiving N alerts *above* a threshold before sending a Notification. See Figure 7. Let’s take a closer look at each type of Notification in turn.

Email: Email is pretty self-explanatory; it simply sends an email using the Mail::Internet modules to the SMTP server you specify or to the Panic destination if that gateway is unreachable. To configure email notifications, use lines like these in your server config file:

```
[global]
alert_type = email
alert_dest =
  redalert-admin@ops.domain.com
alert_host =
        mailhost.ops.domain.com
[category_type]
alert_dest =
  category-owner@eng.domain.com
```

The implication here is true: the definitions in the [global] section will act as defaults unless they’re overridden by category-specific values. This is known as ‘‘translucency’’ in OO parlance.

SNMP: SNMP Notifications are (by the protocol’s very nature) more involved than those of the Email type. They are sent as an SNMP trap to the trap destination port (generally udp/162) of your network management station. The usual object ID of the trap is 1.3.6.1.4.1.43.33.3.9.6, which translates into the ‘‘enterprises.a3Com.palm-mib.redalert’’ branch of the root SNMP-mib2 tree. The last two places are the leaf nodes of the RedAlert MIB: ‘‘traps.sendTrap’’, the value of which trap will look like Listing 5.

If you want to use SNMP traps, you’ll first need some kind of network monitoring software like HP OpenView which first, has the ability receive the traps, and second, can be configured to do something useful, like log a message or flash an icon on the screen, upon receipt of said trap. You should compile the included redalert.asn1 MIB definition into your management station’s MIB tree and prepare its configuration to receive RedAlert traps (the procedure varies on the software you’re using).

In the server’s config file, setting ‘‘alert_type’’ to ‘‘snmp’’ requires you to add an ‘‘alert_community’’ variable and transform the meaning of the ‘‘alert_dest’’ line from an email address into an SNMP Object-ID with which to tag the trap. Again, lines in the [global] section will be used as defaults unless you override them in the section for a Category.

```
[global]
alert_type      = snmp
alert_host      =
          traphost.ops.domain.com
alert_dest      =
          1.3.6.1.4.1.43.33.3.9.6
alert_community = public
[category_type]
alert_dest      =
         otherhost.ops.domain.com
alert_community = private
```

RedAlert uses the SNMP::Session [5] module to build up the UDP trap packet, encoding all of these values and adding the Notification message, and then registers it as an Event in the main Event Loop to send as soon as possible. This generates a ‘‘queue’’ of outgoing SNMP alerts, which if not sent within a ten-second timeout, will go into ‘‘Panic’’ mode and send the notification (plus the fact that it couldn’t be sent) to your Host of Last Resort.

OTHER: RedAlert is always evolving and maturing, and among the ways it is becoming more useful are additional Notification messages. A Syslog facility is in the works, further refinements might add the ability to dial a modem and speak TAP directly to an alphanumeric paging service. The simple, standard, text-based nature of the Notifications make it very easy to add new backends in response to new technologies or your site-specific requirements.

Gilding the Lily

As a generic Perl program, a RedAlert client can of course undertake whatever additional, non-RedAlert actions you require, such as logging the

```
1 Panic! Couldn’t connect to traphost.ops.domain.com sending:
2 Trigger threshold exceeded:
3 Cannot fork on operation SMTP_VRFY! Message-ID:
                  <199904291241.FAA17062@mailhost.ops.domain.com>
```

Listing 4: Worst-case notification.

```
Trigger threshold exceeded:
Cannot fork on operation SMTP_VRFY! Message-ID:
      <199904291241.FAA17062@mailhost.ops.domain.com>
```

Listing 5: The traps.sendTrap.

message-ids of "hostname not found" bounce mail. An unusually diligent postmaster whose queues are on a Network Appliance server might save the message-ids and try to pull back the queued messages via snapshot files, but this level of hand-intervention is certainly uncommon in an ISP environment.

An individual running their own site could easily attempt to resubmit bounced mail of certain types using a modified RedAlert client. It would require relatively little effort to write a client which could tally mail bounces for you and compare each bounced recipient and destination against a personal directory of frequent addressees, rewriting and resubmitting such bounces that appear to be simple typos on your part. Postmaster copies of bounces will usually have the body deleted for privacy/security reasons, so this approach works only with the individual recipient copies.

If you are running high-volume services and wish to do sophisticated post-processing of messages, such as rewriting and resubmission, you will probably wish to redirect them to a separate server. Since most filtering methods are non-threaded, and many have clumsy locking, separating the traffic from your important mail server is a reasonable idea.

Extensions for the Future

- Filter messages right in the MTA and call our Sendmail client as a program or local mailer.
- Object Persistence in the form of a database backend to the RedAlert server. This would allow greater reporting and analysis of incoming Alerts, as well as enabling Threshold state to be maintained indefinitely.
- Add SNMP listener to make internal state of client monitoring and configuration available directly.

Availability

The RedAlert code base is released under the GNU Public License, and is available via FTP or by anonymous CVS. If you're interested in using RedAlert, or even in helping to make it more useful by joining the development team, check out the RedAlert homepage at: <URL:http://explosive.net/opensource/redalert/>

The extreme configurability of the client side and the robust, platform independent nature of Perl lead us to believe that the tool will find wide acceptance and use in the systems administration community. Email and Squid monitoring clients exist as of this writing, and we look forward with interest to seeing new clients emerge. For general RedAlert discussion, please send mail to majordomo@explosive.net with "subscribe redalert" in the subject of your message. You will automatically be sent back an authentication token to complete the subscription process, so please make sure your return address is valid.

Acknowledgements

As with any open source project, we stand on the shoulders of giants and add our small contribution to the view. We would like to thank Larry Wall and all the CPAN contributors for their excellent and irreplaceable work in creating the foundation for RedAlert and countless other tools. We also express our gratitude to the pioneers of SNMP, and to 3Com for granting us space in the Enterprise MIB.

Author Information

Eric Sorenson is a UNIX sysadmin and systems programmer currently living the good life in Silicon Valley. When he's not LARTing developers or working on his hardened garage NOC, he enjoys ultralight sailboat racing with his friends and fiancee and listening to experimental ambient music. Reach him electronically at <eric@explosive.net>.

Strata Rose Chalup <strata@virtual.net> has been a sysadmin professionally since 1983, long before all the Internet hype. For the past several years, she has specialized in high-volume network service architecture, particularly for email. Following a road paved with good intentions, Strata has drifted into the shady netherworld of Project Management and looks forward to resuming more hands-on technical work. All year she has been promising herself "just one more contract" before taking a month off to catch up on scuba diving, learning Perl, and getting back into tai chi. In her minimal spare time, Strata is learning to program her new digital camera and working with her husband to convert a 24-foot school bus into a custom motorome for dive weekends up and down the California coast.

Inline References

[1] Data::Dumper 2.101 by Gurusamy Sarathy, <gsar@umich.edu>.

[2] MailTools 1.13 by Graham Barr, <gbarr@pobox.com>.

[3] AppConfig 1.52 by Andy Whardley, <abw@cre.canon.co.uk>.

[4] Event 0.51 by Joshua Pritikin, <bitset@mindspring.com>.

[5] SNMP_Session 0.70 by Simon Leinen <simon@switch.ch>.

Other References

Chalup, Hogan, et al., "Drinking From the Fire Hose," 1998 LISA.

Pree, Wolfgang, "Design Patterns for Object-Oriented Software Development," 1995, ACM Press.

Friedl, Jeffrey E. F. "Mastering Regular Expressions," 1997, O'Reilley and Associates.

Srinivasam, Srinam. *Advanced Perl Programming*, 1997, O'Reilley and Associates.

Comprehensive Perl Archive Network (CPAN),
AppConfig-1.52 by Andy Whardley <abw@cre.canon.co.uk>,
Data::Dumper-2.101 by Gurusamy Sarathy <gsra@umich.edu>,
Event-0.51 by Joshua Pritikin <joshua.pritikin@db.com>,
Event-tcp-0.007 by Joshua Pritikin <joshua.pritikin@db.com>,
MailTools-1.53 by Graham Barr <gbarr@pobox.com>,
SNMP_Session-0.70 by Simon Leinen <simon@switch.ch>.