Fixing On-call, or How to Sleep Through the Night

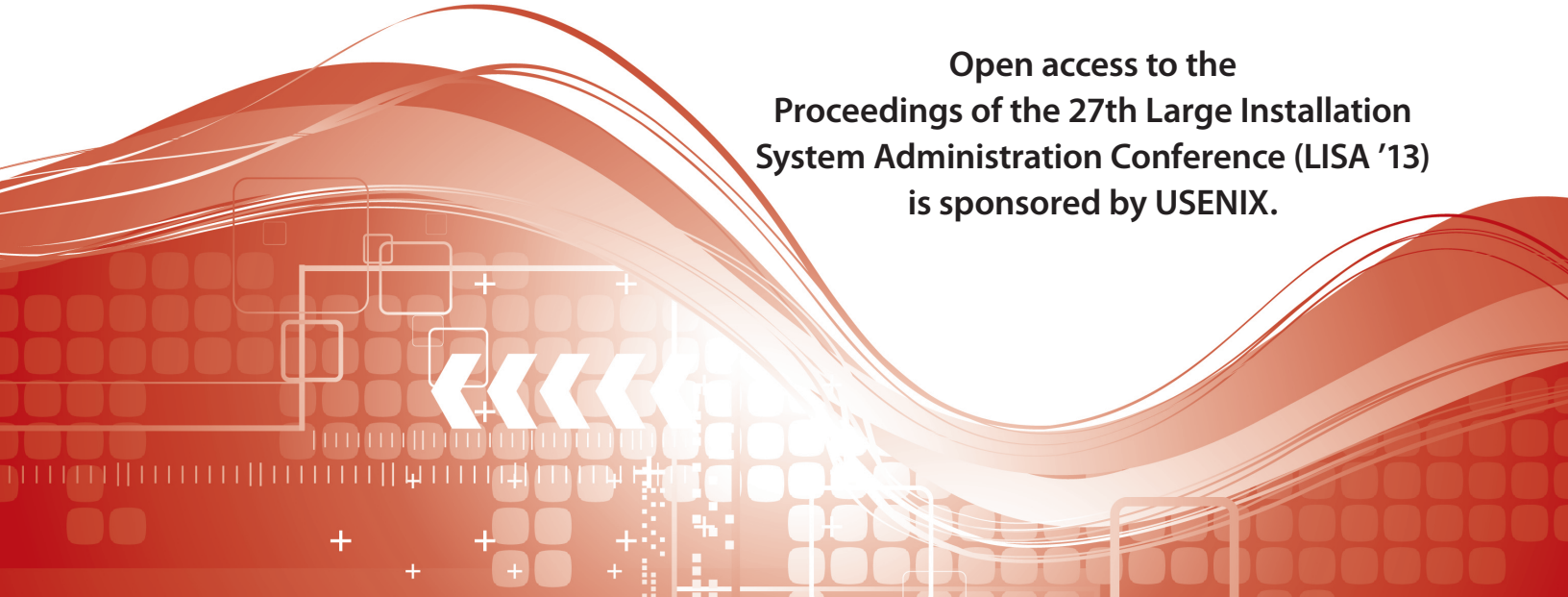
Matt Provost, *Weta Digital*

https://www.usenix.org/conference/lisa13/technical-sessions/papers/provost

This paper is included in the Proceedings of the 27th Large Installation System Administration Conference (LISA '13).

November 3–8, 2013 • Washington, D.C., USA

ISBN 978-1-931971-05-8



Open access to the Proceedings of the 27th Large Installation System Administration Conference (LISA '13) is sponsored by USENIX.

Matt Provost
Weta Digital
Wellington, New Zealand
mprovost@wetafx.co.nz

Tags: Nagios, monitoring, alerting

Fixing On-call, or How to Sleep Through the Night

ABSTRACT

Monitoring systems are some of the most critical pieces of infrastructure for a systems administration team. They can also be a major cause of sleepless nights and lost weekends for the on-call sysadmin. This paper looks at a mature Nagios system that has been in continuous use for seven years with the same team of sysadmins. By 2012 it had grown into something that was causing significant disruption for the team and there was a major push to reform it into something more reasonable. We look at how a reduction in after hour alerts was achieved, together with an increase in overall reliability, and what lessons were learned from this effort.

BACKGROUND

Weta Digital is Sir Peter Jackson's visual effects facility in Wellington, New Zealand. Because of the workflow at Weta, users are at work during the day but our 49,000 core renderwall is busiest overnight when none of the sysadmins are around. So often the highest load at the facility is overnight which is when problems are most likely to occur.

The monitoring system at Weta Digital uses the open source Nagios program and was introduced into the environment in January 2006 following the release of "King Kong". Nagios was replaced with the forked project Icinga in September 2010 but the core of the system has remained the same.

The Systems team at Weta is on a weekly on-call rotation, with 6-8 admins in the schedule. However some admins had configured specific systems to page them directly, and some alerts were configured in Nagios to always go to a specific person, possibly in addition to the on-call pager. So some sysadmins were essentially permanently on-call.

By 2010 the system had become unsustainable. The weekly on-call rotation was dreaded by the sysadmins. Being woken up every night while on-call was not unusual, in fact getting a good night's sleep was remarkable. Being woken up several times during the night was so common that we had to institute a new policy allowing the on-call sysadmin to sleep in and come to work late when that happened. However many (if not most) of the alerts overnight were non-critical and in many cases the admin would get the alert and then go back to sleep and wait for it to recover on its own, or wait to fix it in the morning.

Some of the admins were starting to compensate for bad on-call weeks by leaving as early as lunchtime on the following Monday once they were off call. So the average number of hours across the two weeks of on-call and the next week still averaged out to two normal weeks, but it was impacting on their project work by essentially missing a day of work and often left the team shorthanded when they were out recovering from on-call.

Sysadmins at Weta are independent contractors on an hourly rate. Each after hours alert therefore creates a billable hour. It could be construed that there is a financial incentive for creating new alerts, whether people think about it that way or not. The team itself is responsible for maintaining the Nagios system and adding alerts, so asking them to try and reduce the number of alerts, thereby reducing their income, was always going to be an unpopular move.

NAGIOS

One of the differences with the Weta system is that it uses a set of configuration files that are processed by a Perl script to generate a Nagios configuration. This is used to manage the most common configuration changes via a single file, hosts.txt, which manages both host and service checks. The hosts.txt file is similar in format to the original Netsaint configuration. (Netsaint was the original name of the Nagios software in 1999 but was changed in 2005 for trademark reasons.) The original Netsaint configuration files consisted of lines with semicolon delimited fields:

Format:

```
host[<host_name>]=<host_alias>;<address>;<parent_hosts>;<host_check_command>;<
max_attempts>;<notification_interval>;<notification_period>;<notify_recovery>;
<notify_down>;<notify_unreachable>;<event_handler>
```

Example:

```
host[es-gra]=ES-GRA Server;192.168.0.1;;check-host-alive;3;120;24x7;1;1;1;
```

The Weta configuration files look similar, but with colons separating the fields:

Format:

```
<host_name>:<host_type>:<check_name>:<check_command>:<parent_host>
```

Examples:

```
imap::check_long_running_procs,nosms:check_long_running_procs:vrrpe120
adsk-license2:Linux:check_autodesk_is_running:passive:vrrpe120
ldapauth0:Linux:ldap:check_ldap!ou=People,dc=wetafx,dc=co,dc=nz:vrrpe120
```

Because the config is reduced to one line per command it is trivial to add large new sections of checks with simple for loops. This contributed in part to the proliferation of checks in the configuration. In some cases there are specific scripts that are used to generate hundreds of checks.

Most of the specifics for each host and service check are filled in via the Perl script using template files. The config is stored in RCS which has allowed us to review and analyse the history of the configuration since 2006.

THE PROBLEM

Over the years the amount of hosts being monitored by Nagios has grown from an initial 297 to 843 today, with a peak of 915 in 2011. The number of service checks has grown at a much higher rate, from 365 initially to 3235 today (April 2013), with a peak of 3426 in December 2012.

How did the number of checks grow so much over time? For the most part it was just the natural accumulation of new checks being added, and then retroactively being added to other hosts. So adding a new check to a Linux host that then gets applied to all of the others can easily add hundreds of checks. There were a few incidents like this where there were step changes and many checks were added, but there was also just the slow growth of adding servers. Moving much of our server infrastructure to VMware has certainly increased the number of individual hosts in use. Of course over time there are also an increasing number of legacy systems to maintain.

Starting in October 2010, in response to a growing number of complaints from the team about on-call, I began a project to reduce the number of checks that would alert the Systems team, by increasing the number of alerts that were configured not to send pages. However nothing was done to reduce the number of checks themselves. The number of email problem notifications was still averaging over 50 per day. In fact there were so many alerts that were being ignored that some started to be labelled with the suffix '_fix_asap' to distinguish them from all of the noise. That was a critical sign that the system was broken. People were configuring Nagios to alert for 'their' systems to make sure that they were being looked after. But this meant that some systems were being aggressively alerted, more than necessary.

Ignored alerts are false alarms and are Nagios itself notifying us that it was really the system that was broken. And based on that, our most unreliable system was Nagios itself. So it needed a major overhaul. The amount of alerts going to the on-call sysadmin and the number of emailed alerts going to the whole team were still overwhelming, so in January 2012 I started another project for the team to clean up the Nagios configuration.

CLEANING UP

One of the features of the original Perl template system is called 'nosms'. This allows the admin to specify that (originally) a particular host or (later) a service check should not send an SMS alert to the on-call sysadmin. This was designed to keep non-mission-critical systems from alerting anyone out of hours.

However adoption of the nosms option was very low - it started with 2 hosts in January 2006 and was only up to 30 by January 2011. Under pressure to improve the on-call situation it had increased to 202 a year later in 2012, but with a concerted effort it is now at 807 checks. Along with other changes to the system to stop overnight alerts, the number of checks that can wake someone up is down to 1280 from a high of 1763. That number is still too high and work is ongoing to reduce it further.

Individual members of the team had also configured Nagios to page them specifically when 'their' systems were having issues, even when they weren't on-call. These have all been removed, so now when you are not on-call you won't be bothered. Even though people were doing this to be helpful and look after the systems for which they are responsible, it just creates burnout for those individuals. The basic rule now is, when you're not on-call, you don't get paged.

I also instituted a new policy, that the on-call sysadmin has to call the vendor before waking up someone else on the team. Due to the small size of the team and the large number of different systems that we have to support there is some inevitable siloing within the team where certain people look after certain systems. The tendency is to call those people first when 'their' system is in trouble, but that just penalises an admin who has to look after the more unreliable systems, even if there is nothing that they can do to improve the situation. Some hardware is just less reliable than others, and unfortunately you don't often discover that before you buy it. This was creating an environment where people were afraid to express interest in and thereby become responsible for some new equipment in case it turned out to be unreliable. Weta pays a significant amount of money each year for support contracts so that we can call tech support 24/7. So the new rule places more of a burden on the vendors and less on other team members.

One change that we made as a team was to move to a consensus based approach to adding new alerts. So new alerts have to be discussed and if even one person doesn't think that it's important enough to wake up for then it isn't added to the configuration. Whoever wants to add the alert has to convince everyone else on the team that it is worth waking up for. This stops people from adding extra alerts for specific systems without consulting with the rest of the team. It's important to do this in an open, non-confrontational way, preferably in person and not over email where discussions can get heated. It has to be a positive environment where senior members of the team can't intimidate more junior members to go along with their ideas. Management also has to provide input and get feedback from the stakeholders of a particular system to make sure that business requirements are still being met.

Nagios shouldn't be used as a means of avoiding work in the future. Priority needs to be placed on not bothering the on-call sysadmin with problems that can be deferred until the next working day. In some cases systems could be configured to run overnight but it would create a mess that someone would have to clean up in the morning. There is a natural tendency for the person who will most likely have to clean things up to make an alert and wake someone up to deal with the problem before it gets severe and creates a situation where there would be more effort involved to clean it up after the fact. In that case there is only a 1/8 chance that that person responsible for that system would be the one on-call, but they would always have to clean things up in the morning. So the natural reaction was to create an alert. Instead we should be placing a higher importance on respecting people's time (and sleep) when they are on-call so doing more work during office hours is preferable to alerting someone out of hours.

In our Nagios configuration some systems were identified by multiple DNS CNAMEs. This would happen because we often separate the system's hardware name from a more functional name, but then both would be monitored. This would result in duplicate alerts, although it usually wasn't until you logged in and checked that you would realise that they were for the same system. It also created confusion during more critical events like a network outage where Nagios wasn't correctly reporting the actual number of hosts that were affected. So we went through a cleanup and rationalised all of the hostnames down to a unique set.

THE LIST

By January 2012 tensions around on-call reached a breaking point. I decided that my earlier efforts to gradually clean up the system were happening too slowly - for every alert that was being nosms'd or deleted, two more new alerts were created. I began by consulting with the CIO, who was shocked at the amount of monitoring email alerts in her mailbox, and coming up with a list of systems that were deemed critical to the facility and should be monitored 24/7. This list was quite small in comparison - less than 50 hosts as compared to the over 1600 in Nagios at the time.

This led to a time period that I call the 'Constitutional Phase' because of how the list was pored over for inconsistencies by the team as if it was a legal document. Faced with an extremely shortened list of hosts, the team started to question each decision as to why certain hosts were on or off the list. For example, "If host A is on the list, host B is just as important so it should be on there as well." Or "If host C isn't on the list then D and E shouldn't be on there either."

Rather than trying to articulate and justify each decision individually I took a new approach which was to try and come up with a flow chart or decision tree that would have a consistent set of questions to determine whether each host in the Nagios config would continue to alert or not, eventually resulting in the much shortened list. One example of such a list of questions was:

Is it a critical production system?
  No dev/test/stage
  No backup/offline/nearline
Is it the renderwall?
  Will it affect more than 50% of the renderwall?
Is it a critical piece of the Production pipeline?
  Dailies/Filemaker/email
Will it affect clients?
  Aspera/SFTP/external website/external email
Will it affect all users of a piece of core render software?
  license servers

However in an attempt to try and limit the list to things that would be considered major outages, I was falling into the trap of setting thresholds (will it affect more than 50% of the render servers). So I was never very happy with this approach and it was never implemented.

The next attempt was to focus on who was affected:

Will it affect:
  Dailies?
    Core render software for the renderwall
  Users?
    Core software for users/email/Filemaker
  Clients?
    Aspera/SFTP/email/website

But this would lead to keeping almost everything on the list because almost every system affects renders, users or clients. So again it was never implemented.

The final approach that I took was to simplify the list of questions to the most basic ones:

Is it:
A production system (no dev/test/stage) and
A primary system (no backup/offline/nearline)
or a single point of failure for another critical system

This does pretty much lead to an unchanged list with the exception of removing things like dev systems (yes, before this dev and test systems would normally alert). However there was a fourth question that I added which changed the whole focus of the project:

Is it configured to automatically reboot/restart/repair itself?

I realised that focusing on Nagios itself wasn't the goal. Only a certain amount of non-critical systems could be either removed or set not to alert after hours, but we weren't really fixing anything but Nagios itself. The number of alerts coming in wasn't really changing, it was just that fewer of them were going to the on-call person's phone. The team's workflow consisted of finding a problem and then adding a Nagios alert for it, and never actually going back and resolving the underlying issue. Once it was in Nagios it was considered 'fixed' even though it kept alerting and was being resolved by hand. We needed to change our focus from just monitoring systems to actually fixing them.

Once I had a new list, which was actually now just a set of questions, I ran it past the CIO to make sure that I had her support before continuing. It was important to get her approval to make sure that it would meet the business' needs, and she agreed with the direction that the project was taking, even allowing some decisions which the team was convinced would lead to reduced service levels. However, she highlighted the need for striking the right balance through trial and error.

FIXING

Nagios was being used as a job scheduler to get sysadmins to perform some task. Alerting had become the culture of the team and was viewed as the solution to every problem. Where possible all of those tasks should be automated instead.

As an example, we had Nagios alerts set up to tell us when there were security updates available for servers in our DMZ network. Since they were all running the same version of Debian you would get a flood of identical alerts at the same time as a new update became available. This was configured as a nosms alert, but someone on the team would see the emails and manually install the updates. This has now been replaced with apticron which downloads and installs the updates automatically. The Nagios check remains to alert us when this process fails and human intervention is still required.

Before 2009 there was nothing configured to automatically restart failed programs. Starting in 2009, after a string of alerts from an unreliable daemon, we began using DJB's daemontools to automatically restart critical services if they crashed. It's now a rule that we don't monitor anything unless it's configured to automatically restart, where possible. This may seem counterintuitive - we only monitor services that have already been configured to restart. But this places the emphasis on fixing the problem and not just putting in a Nagios check and letting someone deal with it at a later date.

To resolve unresponsive (hung) Linux servers, we first started configuring the kernel to reboot after a panic. By default the system will wait with the panic message, but we're just interested in getting it back up and running. Later in 2012 the kernel watchdog was also enabled. This will automatically reboot a server if the kernel stops responding. In the case where a server is hung there is typically nothing for the on-call person to do but use the out-of-band management to hard reboot the system. Now this happens automatically before a Nagios alert is sent out. For our

servers in VMware there is a monitoring setting available that will automatically restart servers if they are unresponsive and not doing IO.

Weta has many Netapp filers for NFS file serving and sometimes they panic due to software bugs. They come in redundant pairs, but the on-call admin would be notified that one node was down and would log in and reboot it and fail back the cluster. There is a configuration option for automatic cluster giveback which we have enabled on all of our filers now, so unless it's a hardware fault the cluster can automatically recover itself and not alert.

We were also managing disk space manually on the filers before this project. So an alert would be sent notifying us that some filesystem was getting close to capacity, and the on-call sysadmin would log in and extend the filesystem (if possible) or delete some files. We have enabled the filesystem autosize feature where the Netapp will automatically add space to filesystems as they get close to filling up, up to a specified limit. Now the on-call person only gets paged when it hits that limit after expanding several times automatically. This has eliminated most of the overnight disk space management.

Logfiles filling disks was another thing that would wake people up. Now that we're focusing on fixing the alerts, automatic log rotation is set up to stop disks from filling up overnight instead of manually deleting old log files in the middle of the night. Any alert coming in from a log that hasn't been set to rotate should be resolved by adding it to the logrotate configuration.

If you buy a redundant system you should be able to trust it. So we've stopped alerting on failed disks in our filers, and even failed heads since they're part of a redundant clustered pair. We always configure the filers with plenty of hot spare disks. The same rule goes for systems with redundant power supplies or anything else that is designed to keep functioning when one component fails. You are taking the risk that possibly a second component will fail overnight but in practise that is extremely unlikely. One of the reasons for buying redundant hardware is to keep your admin team from being bothered by every failure if the system can keep running.

In the case of load balanced services, instead of alerting on each server behind the load balancer, the Nagios checks have to be rewritten to do an end-to-end check of the service. It doesn't matter if one server fails since the load balancer will hide that from clients. As long as one server is up and running, no alerts need to be sent. If the load is too great for the remaining servers then the service check should timeout or fail and alert the on-call sysadmin as usual.

One phenomenon that I was seeing I call 'Twikiscript' after the wiki that we use for team documentation. There is a page 'NagiosAlertsAndErrorsAndWhatToDo' which contains sections for about 150 different Nagios alerts. Many of them contain instructions that are in the form of "log in, look for this and do this to resolve it". Some of the sections include branches and conditionals, and in essence are scripts written for humans to execute. Most of them could be rewritten as scripts for computers to run with a little work.

For example, one alert that used to reliably wake people up several times a week was sent when our MySQL replication slaves would start to lag behind the master. This could be for a variety of reasons, typically high load or bad queries. The fix, following the instructions in the wiki, was to log in and update the parameter in MySQL that would change the InnoDB engine from the standard behaviour of flushing data to disk after each transaction to only doing it once per second. We call this 'turbonating' the server. This creates the risk that you can lose up to a second's worth of data but also increases performance and typically allows the slaves to catch up. Because they were slave copies, the DBA team was ok with us taking this risk. Often you would get woken up, change this setting, go back to sleep, wait for the recovery alert and then change it back. Now we've created a cron job that watches the lag every 5 minutes and if it's beyond a threshold in seconds it changes the MySQL parameter to the faster setting. If the lag is below the threshold it changes it back to the safer setting. It also emails out so that the DBA team will know that it has happened and investigate possible causes. This is called 'autoturbonation' and since it has gone in has completely eliminated the overnight alerts.

We've also started to use the swatch program to help fix problems automatically. It's a Perl program that tails a log file and searches each line for regular expressions and then can execute a script when it finds a match. It can be used

for notification, so we configure it to email the team when certain errors appear. We also use it to fix things automatically, so when a recurring error shows up that has a known fix, we can script that fix, hook it into the swatch configuration and stop getting alerts.

Alerts should be actionable, and in a reasonable timeframe. We monitor SSL certificate expiry in Nagios, with a 30 day warning to give us time to order a new cert. But typically what happens is that the alert goes off daily for about 28 days and then someone will get around to it.

Another example of non-actionable alerts are the checks for batteries on RAID controllers, which expire after a certain amount of time. This is usually known in advance (they have a service life of a year in most cases) but we use Nagios to alert us when the controller shows an error condition due to the battery expiring. But then an order has to be placed with the manufacturer so it can take weeks to replace a battery. This isn't a good use of an alerting system. Ideally they would be replaced on a fixed schedule which would be maintained in a separate calendaring system, along with reminders for renewing SSL certificates with a fallback alert that goes off shortly before they expire.

One way that we keep everyone focused on fixing Nagios errors is to meet weekly on Mondays to go over all of the Nagios emails from the previous week. That week's on-call sysadmin leads the meeting and goes through each alert and what the response from the team was. In many cases there was no response and the system recovered on its own, so we discuss whether we can change the thresholds in Nagios to avoid being notified in the future. Or if it's a recurring issue that we see happening week after week someone can be assigned the task of fixing the underlying issue. With the Nagios configuration in RCS we can also review any commits made during the past week and make sure that the entire team is ok with those changes. If there are any objections the change can be reverted. This meeting also serves as a critical safety check to ensure that we aren't changing the system too much and drifting over the line where we're not picking up events that are critical to the facility by providing an opportunity for anyone to raise problems with the monitoring system.

THE RESULT

As a result of all of these changes, the number of out of hours alerts for the on-call sysadmin dropped significantly. During the production of "Avatar" and "Tintin", people were sometimes working over 70 hour weeks and doing over 60 hours was common when on-call. During the production of "The Hobbit" last year, noone on the team worked over 60 hours due to being on-call. Now sleeping through the night when on-call is the norm, and getting woken up an unusual situation. And as time goes on it is less and less tolerated by the team so there is an acceleration to the process of eliminating recurring alerts. This has dramatically improved the work/life balance of the sysadmin team by reducing out of hours alerts.

The level of the service provided to the facility didn't decrease, in fact the system is the most stable it has ever been. I attribute at least part of this to the work that was done in trying to reduce alerts by fixing long standing problems and configuring them to fix themselves whenever possible instead of waiting for systems to break and reacting to alerts. Computers can respond much quicker than humans and can often correct problems faster than users notice the problem in the first place.

As the CIO has acknowledged, "Ultimately, major cultural and working-practice changes were involved; a shift of emphasis to proactive and preventative maintenance and monitoring, rather than reactive problem solving; greater teamwork and shared ownership of systems rather than solo responsibility for them; and the adoption of a dev-ops approach to system administration with a greater emphasis on automation, standardisation and continuous improvement. Although it is taking some time and tenacity to get there, the results have been overwhelmingly positive and beneficial."

THE FUTURE

One of the improvements that could be made to simplify the Nagios configuration would be to split it into two systems - one for sending pages to the on-call sysadmin and another for the non-critical alerts. This has been accomplished with the nosms feature of our current infrastructure but it adds unnecessary complexity.

We are looking at integrating Nagios into our ticketing system, RT. This would involve opening a ticket for each Nagios alert and closing the issue when Nagios sends the OK. Any work done to resolve the issue can be recorded against the ticket. This would let us track which issues are being worked on by the team and which are resolving themselves (false alarms) so that we can work on eliminating them.

We're starting to use pynag, a Python module for parsing Nagios configurations, to monitor Nagios itself. For example, we can use it to resolve the hostnames in the configuration and alert us when there are duplicate CNAMEs. Or it can verify that no hostnames ending in '-dev' are set to alert. By setting up rules in code it stops the configuration drifting from the policies that have been set by the team.

The final frontier is integrating the Nagios configuration into our Puppet rules when we configure servers. Then we can generate a configuration end to end. So for example we could specify a service in Puppet and configure it to run under daemontools and then automatically add that service to Nagios. If it isn't configured in Puppet then it won't be added to monitoring. This way we can enforce that it is configured to automatically restart.

Monitoring systems are good at maintaining a list of things to check. In the early days implementing one can be a big improvement over being surprised by finding that critical systems are down or in trouble. But in a complex system you end up enumerating every possible component in the monitoring system which is an almost endless list as you chase the long tail of increasingly unlikely conditions to monitor. It's a reactive process where checks are only added as things break. In the long run the only way out is to approach monitoring from a more systemic point of view, looking at the infrastructure as a whole as opposed to a group of component parts.

For example, in Weta's case, we still have never monitored renders, which are the core function of our compute infrastructure. So every render could encounter some condition that causes it to fail overnight and noone would have any idea until the next morning, if it's something that we haven't accounted for in our Nagios checks. A better check would be to have an automated program reading the logs coming off the renderwall and alerting when something out of the ordinary was going on. At that point Nagios could be used by the sysadmin team to triage and try to narrow down where the error was coming from. But in the long run it should be less and less a part of everyday life for a system administrator.

LESSONS LEARNED

Anytime you go beyond the capabilities of your monitoring system I think you have to take a good look at what you're doing and question whether it is truly necessary. We wrote a front end to the Nagios configuration which made it easier to add new checks, and then proceeded to add so many that we were overwhelmed. And we built the nosms feature instead of realising that Nagios is for alerting and that moving non critical checks into another system altogether would be better. In fact we had to make policies for people to be able to sleep in instead of questioning why they weren't getting any sleep in the first place and whether that sacrifice was worth it. Building a custom monitoring system is probably not the best use of a team's time when working on fixing problems can lead to better results.

Unfortunately if there is money involved then no matter how much individuals might complain about being on-call there is always a financial incentive to work more out of hours. Different people on the team have different tolerances for this based on their personal and family situations which can create tension within the team. Trying to fix the problem of on-call can be perceived as an attack on their paycheck and there is no financial incentive for the team to clean up the broken system so it takes some strong leadership from management initially to make it happen. My experience is that once the changes have been made people start to appreciate that it makes a positive improvement in their personal lives and the focus on money fades.

Monitoring systems can grow over time to cover every possible part of the infrastructure, out of a drive from the team to keep everything running all the time, or as reactions to individual system failures that weren't caught by an alert and were reported by users. However this can lead to a 'tree falling in the forest with noone around' situation where sysadmins get alerted over the weekend about some system that wouldn't be noticed until Monday morning. If it's not mission critical it is possible to step down from that level of service and wait for a user to complain instead of proactively fixing every problem. This takes good backing from management so that team members don't feel individually responsible for these decisions. But in our experience it has worked out well for both users and sysadmins.

Because monitoring is a way of assigning tasks to the team, it is a backdoor way for members of the team to work around whatever management systems you may have in place for assigning daily jobs. Moving to an open, consensus-based approach for maintaining the monitoring configuration makes sure that everything is done the way management wants to set the team's priorities.

Some people would rather fix things before they turn into bigger problems and don't want to defer the work until later. If you want to preserve their sanity and work/life balance there has to be a focus on getting things done during work hours and building systems that can run overnight or for a weekend without any human intervention. Each call that someone gets should be a unique problem brought about by some unexpected circumstance or failure. Any routine alert should be fixed with urgency so that it doesn't further bother people outside of work hours.

SUMMARY

- Get management/stakeholder buy in
- No personal alerts (when you're not on-call you don't get paged)
- Don't optimise alert configuration (no custom scripts to make alerting easier, a little friction is good)
- Keep the config(s) in version control for easy tracking and review
- No priority labels - every alert should be critical (no _fix_asap...)
- Split monitoring system into critical and noncritical (at least by email address)
- Call vendors before bothering teammates
- Consensus based process for adding new alerts (can be done retrospectively in the weekly meeting)
- Prioritise in hours work over after hours (even if it will take longer)
- No CNAMEs/duplicate hosts
- Change process checks to process monitors with automatic restarting (daemontools, monit, runit...)
- Automate server updates (apticron, up2date...)
- Reboot after panics (`echo "30" > /proc/sys/kernel/panic`)
- Kernel watchdog (man 8 watchdog, VMware monitoring)
- Netapp filer cluster automatic giveback (`options cf.giveback.auto.enable on`)
- Netapp automatic volume expansion (`vol autosize /vol/myvolume/ -m 30g -i 5g on`)
- Automatic log rotation and compression (logrotate)
- No individual alerts for load balanced servers/services (end to end service checks instead)
- Turn Twikiscript into automated processes
- MySQL autoturbonating (`set global innodb_flush_log_at_trx_commit = 2`)
- swatch for finding problems and scripting fixes
- Ticket system integration
- pynag to monitor the Nagios configuration for policy violations

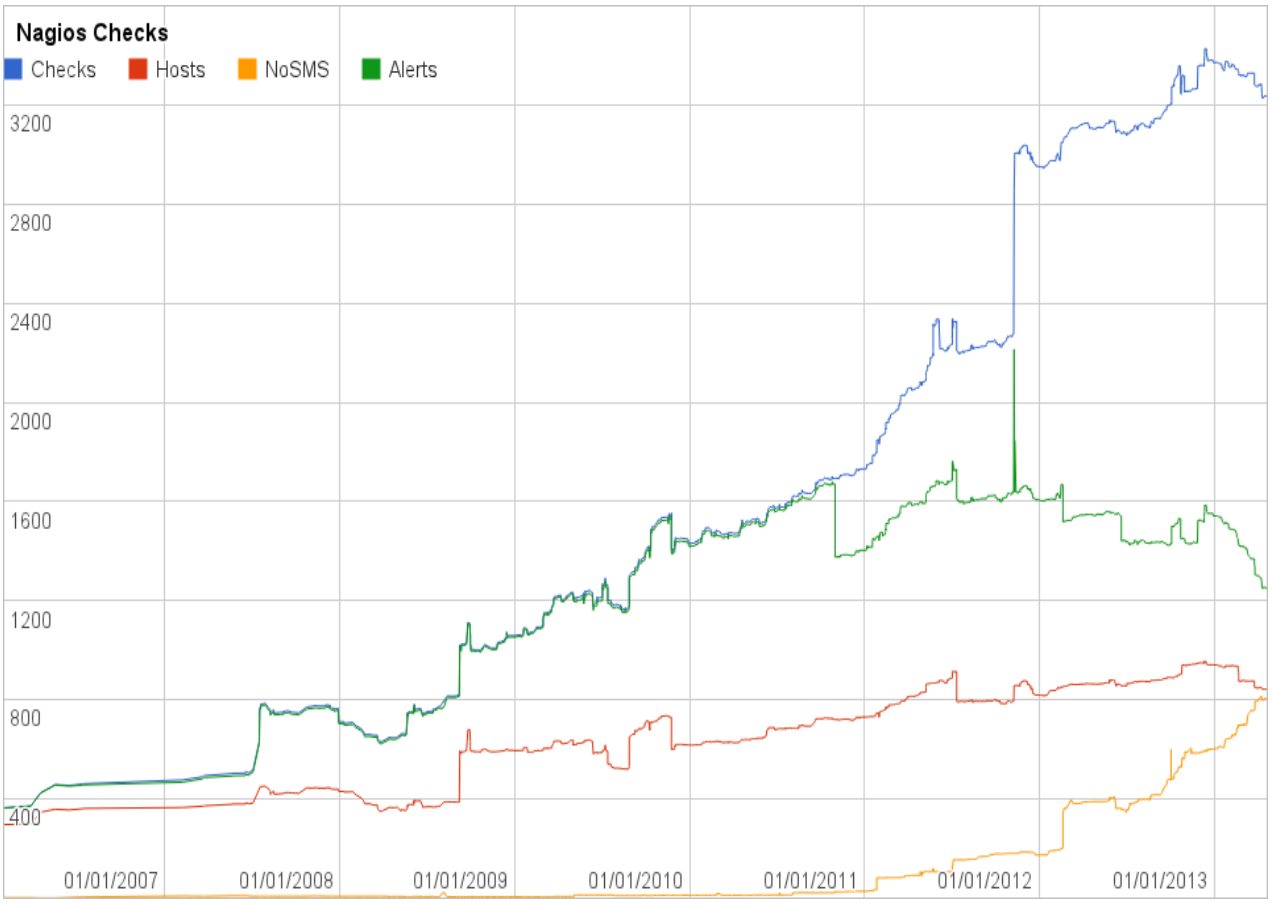


Figure 1. Nagios service checks, hosts being monitored, checks that are configured 'nosms' to not send text alerts, and the total number of alerts going to the team. Note that the total of alerts plus nosms does not total the number of checks since some checks go to other teams.