Nagios and Fruity: What Is Their Monitoring Potential for Your Network?

Jonathan Krein

This article will provide a brief introduction to Brigham Young University’s (BYU) implementation of Nagios and Fruity and present what I have learned about how these monitoring applications perform on both a dedicated server and in a virtual environment. In this article, I’ll describe the limits of Nagios and how its performance varies depending on platform. With the data presented here, you should be better able to project growth potential for a Nagios-Fruity implementation in your own network.

At BYU, the production versions of Nagios and Fruity are housed together on a single dedicated server. In fact, BYU’s network has traditionally been hardware based. However, because of the benefits of virtual servers, our network is beginning to move in the virtual direction. Such a move raises several important questions for Nagios and Fruity:

- Should we move Nagios and Fruity from their dedicated hardware to a virtual environment?
- What would be the ramifications of such a move, and what losses in performance might we incur?
- How much room would Nagios have to grow (i.e., monitoring more hosts and services) on either of these two platforms?
- If we did move to a virtual environment, would we need to distribute Nagios’s load over multiple servers?
- In a distributed environment, how many servers would we need?

To answer these questions, first I will cover some background information about Nagios and Fruity. I will talk about how these applications perform on a dedicated server. Next, I will show what they can do on a virtual server, and I’ll present one server configuration option to be aware of, since it has a large impact on Nagios and Fruity’s performance. I will conclude this discussion with a Nagios load analysis, which should give you a good indication of what Nagios is capable of, particularly in a virtual environment. Along the way, I hope you will also see what potential Nagios has to monitor a large network, as well as the benefits of a configuration file manager like Fruity.

Background Information

Nagios is a Linux-based, enterprise-class monitoring application distributed under an open source license. Nagios works from text configuration files that store information about hosts and services. A host represents a server or other device out on the network, while each service represents a different check that is being performed on its associated host (e.g., CPU utilization, memory usage, processes running). Service checks make up the bulk of Nagios’s load, but hosts are also given a health check when one of their services reports a problem. The real power behind Nagios is that everything Nagios does, from running checks to responding
to alerts, is based on commands that you define. When the status of a host or service changes, Nagios uses these commands to run scripts or other programs, which in turn do the real work. This means that Nagios can do just about anything you can program.

However, defining all of your hosts, services, commands, and other configuration options will create bulky configuration files for Nagios, especially if you have a large network. BYU's implementation currently performs over 2,800 service checks on nearly 1,900 network devices and servers. With so many hosts and services, it’s easy to make a mistake in the configuration files, and a mistake in these files means downtime for Nagios. So, to store all of this data, find configuration errors, and handle the tedious formatting that Nagios requires, we use a configuration file manager called Fruity.

Fruity works from a MySQL database that stores all of the information Nagios needs to do its job. Fruity presents this information for updates, additions, and deletions through a Web interface. One of Fruity’s strengths is that it organizes everything into logical components, thus making it simple to find a host or service, adjust its settings, and then export the changes to Nagios. The best part about Fruity is that it’s also open source and therefore easily obtainable and fully customizable.

**Running Nagios and Fruity on a Dedicated Server**

Figure 1 outlines the dedicated server hardware that has been supporting our production versions of Nagios and Fruity. We have been running these applications on a Linux machine (Fremont) with two processors (hyper-threaded to four) and 4 GB of memory. For most of Fremont’s life, Fruity has performed well, and Nagios’s latency has been under one minute.

The term latency refers to how much later a check runs from the time it was scheduled. Because Nagios works sequentially, it will not start the next check until the previous one has finished. Therefore, latency begins to increase when the load gets too high and checks are scheduled more closely together than they can run.

Our goal at BYU is to keep Nagios’s service check latency below one minute. However, we are now exceeding this limit on Fremont, and the latency continues to increase with each service check we add. Because hosts are only checked when one of their services reports a problem, and these checks are ushered to the front of the queue, host check latency has not yet been affected. Nevertheless, with service check latency now averaging over one minute and Fruity running slower every day, we have been forced to consider distributing the load. As part of this transition, we have been testing Nagios on virtual servers.

**Nagios and Fruity on a Virtual Server**

Figure 2 outlines one of the virtual servers (Brighton) on which we installed Nagios and Fruity. Brighton was first configured to have two virtual processors and a little more than 2 GB of memory. However, even with only 30% of Fremont’s
load, Nagios’s service checks were running behind by more than 6 minutes. Obviously, a virtual server is not going to perform as well as dedicated hardware, but six times the latency on less than a third of the load is far outside of the expected and acceptable performance margins. We further experimented with the optimizations outlined in Nagios’s documentation, but we did not see any improvement until we finally changed Brighton’s virtual server configuration.

**Tweaking the Virtual Server Configuration**

Looking at the graphs that VMware produces on CPU performance, we noticed that each of Brighton’s two virtual processors was only running at 25% capacity; yet Nagios was buckling under its load, and Fruity was running extremely slowly. So we changed the configuration from two virtual processors to one and increased the granted memory to 3.6 GB (see Figure 3). The service check latency promptly dropped to between 0.5 and 15 seconds — almost a 96% reduction — and Fruity accordingly performed better.

**What Happened?**

Take a look at the performance graphs for Brighton in Figure 4. On June 5th, we increased the load on Brighton, and the active memory usage rose to almost 1 GB. However, notice that the CPU performance graph for that same day shows no significant change in processor utilization, which remained at about 50% for both processors (cumulative).

**Could Something Be Holding These Processors Back?**

Look at June 8th, when we made the configuration changes. Active memory did not increase, but the CPU performance rose to almost 70%. We later increased another virtual server’s granted memory without modifying the number of processors and found no changes in CPU performance, active memory, or Nagios’s latency. It seems that on virtual servers managed by VMware, Nagios, and Fruity perform much better with only one virtual processor rather than two. The number of processors is definitely something to keep in mind when setting up these applications, especially in a virtual environment.

**Load Testing the Virtual Environment**

At this point, having configured our virtual server for better performance, we wanted to find out how much load Nagios and Fruity could handle and what the latency curve would look like as we increased the number of service checks. Figure 5 shows a chart of our results — Nagios’s service check latency graphed against the number of service checks being run. Just as we saw with Fremont, Nagios’s latency on Brighton remained low up to a point, after which the checks were being scheduled too closely together, and Nagios would get behind. One interesting thing to note is that the graph is almost linear — a trend that we have also seen with Fremont. Additionally, from the blown-up area of the graph we can see that on a VMware virtual server configured
as outlined in Figure 3, Nagios is capable of about 870 service checks before the latency exceeds one minute.

It is important to note that this graph and the other data presented in this article represent our own experience with the performance of Nagios and Fruity and may differ to some extent from yours. Particularly, keep in mind that your server’s check threshold for Nagios will depend on how long your service checks take to run and how frequently you run them.

This data is most valuable as a model. It clearly shows that a relationship exists between Nagios’s performance and the number of virtual processors on which it is running. It also gives us an idea of what load Nagios and Fruity can handle on various platforms and how Nagios’s latency will increase as load is added. This data should help you answer the questions posed at the beginning of this article and better plan a future for a Nagios-Fruity system in your own network.

Jonathan Krein is a Systems Management Engineer for the Office of Information Technology at Brigham Young University. He is currently working on his Bachelor of Science degree in Computer Science at BYU and can be reached at: JonathanKrein@byu.edu.

---

**Figure 1** Nagios dedicated server specifications and performance

<table>
<thead>
<tr>
<th>NAME:</th>
<th>Fremont</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE:</td>
<td>Dedicated Server</td>
</tr>
<tr>
<td>OS:</td>
<td>Red Hat Enterprise Linux 3</td>
</tr>
</tbody>
</table>

**HARDWARE:**
- Processors: 2x Pentium 4*  
- Memory: 4 GB  
- Hard Disk: 32 GB

**VERSIONING:**
- Nagios: 2.0b5  
- Fruity: 1.7*  
- Apache: 2.0.46  
- MySQL: 4.1.10a  
- PHP: 4.3.2

---

**Figure 2** Nagios virtual server, 1st attempt specifications and performance

<table>
<thead>
<tr>
<th>NAME:</th>
<th>Brighton</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE:</td>
<td>VMWare ESX Server 2.5.2</td>
</tr>
<tr>
<td>OS:</td>
<td>Red Hat Enterprise Linux 3</td>
</tr>
</tbody>
</table>

**DEVICES:**
- Virtual CPUs: 2  
- Memory: 2.048 GB  
- Hard Disk: 25.6 GB

**VERSIONING:**
- Nagios: 2.2  
- Fruity: 1.7*  
- Apache: 2.0.46  
- MySQL: 5.0.18-standard  
- PHP: 5.1.2

---

**Figure 3** Nagios virtual server, 2nd attempt specifications and performance

<table>
<thead>
<tr>
<th>NAME:</th>
<th>Brighton</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE:</td>
<td>VMWare ESX Server 2.5.2</td>
</tr>
<tr>
<td>OS:</td>
<td>Red Hat Enterprise Linux 3</td>
</tr>
</tbody>
</table>

**DEVICES:**
- Virtual CPUs: 2 → 1  
- Memory: 2.048 → 3.6 GB  
- Hard Disk: 25.6 GB

**VERSIONING:**
- Nagios: 2.2  
- Fruity: 1.7*  
- Apache: 2.0.46  
- MySQL: 5.0.18-standard  
- PHP: 5.1.2

---

**Nagios Performance:**

**Active Host Checks:** 328  
**Average Execution Time:** 0.501 sec/check  
**Average Check Latency:** 0.000 sec.

**Active Service Checks:** 861  
**Average Execution Time:** 1.510 sec/check  
**Average Check Latency:** 366.643 sec.

---

* developed at Brigham Young University from Fruity 0.7

* hyper-threaded

**Nagios Performance:**

**Active Host Checks:** 328  
**Average Execution Time:** 0.559 sec/check  
**Average Check Latency:** 0.000 sec.

**Active Service Checks:** 861  
**Average Execution Time:** 2.061 sec/check  
**Average Check Latency:** 14.749 sec.
**Figure 4** Brighton’s CPU and memory performance

![CPU Performance Graph]

- Changed from 2 virtual CPU’s to 1 and increased granted RAM to 3.6 GB.

**Figure 5** Nagios load testing in a virtual environment

![Memory Performance Graph]

- Results: increase in CPU performance; no change in memory usage; latency dropped from ≈6 min. to between 0.5 and 15 seconds.

![Nagios’ Latency vs. Number of Service Checks on Brighton Graph]