CHAPTER 1

Best Practices

Building a monitoring infrastructure is a complex undertaking. The system can potentially interact with every system in the environment, and its users range from the layman to the highly technical. Building the monitoring infrastructure well requires not only considerable systems know-how, but also a global perspective and good people skills.

Most importantly, building monitoring systems also requires a light touch. The most important distinction between good monitoring systems and bad ones is the amount of impact they have on the network environment, in areas such as resource utilization, bandwidth utilization, and security. This first chapter contains a collection of advice gleaned from mailing lists such as nagios-users@lists.sourceforge.net, other systems administrators, and hard-won experience. My hope is that this chapter helps you to make some important design decisions up front, to avoid some common pitfalls, and to ensure that the monitoring system you build becomes a huge asset instead of a huge burden.

A Procedural Approach to Systems Monitoring

Good monitoring systems are not built one script at a time by administrators (admins) in separate silos. Admins create them methodically with the support of their management teams and a clear understanding of the environment—both procedural and computational—within which they operate.

Without a clear understanding of which systems are considered critical, the monitoring initiative is doomed to failure. It's a simple question of context and usually plays out something like this:

Manager: "I need to be added to all the monitoring system alerts."

Admin: "All of them?"

Manager: "Well yes, all of them."

Admin: "Er, ok."

The next day:

Manager: "My pager kept me up all night. What does this all mean?"

Admin: "Well, /var filled up on Server1, and the VPN tunnel to site5 was up and down."

Manager: "Can't you just notify me of the stuff that's an actual problem?"

Admin: "Those *are* actual problems."

Certifications such as HIPAA, Sarbanes-Oxley, and SAS70 require institutions such as universities, hospitals, and corporations to master the procedural aspects of their IT. This has had good consequences, as most organizations of any size today have contingency plans in place, in the event that something bad happens. Disaster recovery, business continuity, and crisis planning ensure that the people in the trenches know what systems are critical to their business, understand the steps to take to protect those systems in times of crisis, or recover them should they be destroyed. These certifications also ensure that management has done due diligence to prevent failures to critical systems; for example, by installing redundant systems or moving tape backups offsite.

For whatever reason, monitoring systems seem to have been left out of this procedural approach to contingency planning. Most monitoring systems come in to the network as a pet project of one or two small tech teams who have a very specific need for them. Often many different teams will employ their own monitoring tools independent of, and oblivious of, other monitoring initiatives going on within the organization. There seems to be no need to involve anyone else. Although this single-purpose approach to systems monitoring may solve an individual's or small group's immediate need, the organization as a whole suffers, and fragile monitoring systems always grow from it.

To understand why, consider that in the absence of a procedurally implemented monitoring framework, hundreds of critically important questions are nearly impossible to answer. For example, consider the following questions.

- What amount of overall bandwidth is used for systems monitoring?
- What routers or other systems are the monitoring tools dependent on?
- Is sensitive information being transmitted in clear text between hosts and the monitoring system?

If it was important enough to write a script to monitor a process, then it's important enough to consider what happens when the system running the script goes down, or when the person who wrote the script leaves and his user ID is disabled. The piecemeal approach is by far the most common way monitoring systems are created, yet the problems that arise from it are too many to be counted.

The core issue in our previous example is that there are no criteria that coherently define what a "problem" is, because these criteria don't exist when the monitoring system has been installed in a vacuum. Our manager felt that he had no visibility into system problems and

when provided with detailed information, still gained nothing of significance. This is why a procedural approach is so important. Before they do anything at all, the people undertaking the monitoring project should understand which systems in the organization are critical to the organization's operational well-being, and what management's expectation is regarding the uptime of those systems.

Given these two things, policy can be formulated that details support and escalation plans. Critical systems should be given priority and their requisite pieces defined. That's not to say that the admin in the example should not be notified when /var is full on Server1;only that when he is notified of it, he has a clear idea of what it means in an organizational context. Does management expect him to fix it now or in the morning? Who else was notified in parallel? What happens if he doesn't respond? This helps the manager, as well. By clearly defining what constitutes a problem, management has some perspective on what types of alerts to ask for and more importantly...when they can go back to sleep.

Smaller organizations, where there may be only a single part-time system administrator (sysadmin), are especially susceptible to piece-meal monitoring pitfalls. Thinking about operational policy in a four-person organization may seem silly, but in small environments, critical system awareness is even more important. When building monitoring systems, always maintain a big-picture outlook. If the monitoring endeavor is successful, it will grow quickly and the well-being of the organization will come to depend on it.

Ideally, a monitoring system should enforce organizational policy rather than merely reflect it. If management expects all problems on Server1 to be looked at within 10 minutes, then the monitoring system should provide the admin with a clear indicator in the message (such as a priority number), a mechanism to acknowledge the alert, and an automatic escalation to someone else at the end of the 10-minute window.

So how do we find out what the critical systems are? Senior management is ultimately responsible for the overall well-being of the organization, so they should be the ones making the call. This is why management buy-in is so vitally important. If you think this is beginning to sound like disaster recovery planning, you're ahead of the curve. Disaster recovery works toward identifying critical systems for the purpose of prioritizing their recovery, and therefore, it is a methodologically identical process to planning a monitoring infrastructure. In fact, if a disaster recovery plan already exists, that's the place to begin. The critical systems have already been identified.

Critical systems, as outlined by senior management, will not be along the lines of "all problems with Server1 should be looked at within 10 minutes." They'll probably be defined as logical entities. For example "Email is critical." So after the critical systems have been identified, the implementers will dissect them one by one, into the parts of which they are composed. Don't just stay at the top; be sure to involve all interested parties. Email administrators will have a good idea of what "email" is composed of and criteria, which, if not met, will mean them rolling their own monitoring tools.

Work with all interested parties to get a solution that works for everyone. Great monitoring systems are grown from collaboration. Where custom monitoring scripts already exist, don't dismiss them; instead, try to incorporate them. Groups tend to trust the tools they're already using, so co-opting those tools usually buys you some support. Nagios is excellent at using external monitoring logic along with its own scheduling and escalation rules.

Processing and Overhead

Monitoring systems necessarily introduce some overhead in the form of network traffic and resource utilization on the monitored hosts. Most monitoring systems typically have a few specific modes of operation, so the capabilities of the system, along with implementation choices, dictate how much, and where, overhead is introduced.

Remote Versus Local Processing

Nagios exports service checking logic into tiny single-purpose programs called *plugins*. This makes it possible to add checks for new types of services quickly and easily, as well as co-opt existing monitoring scripts. This modular approach makes it possible to execute the plugins themselves, either locally on the monitoring server or remotely on the monitored hosts.

Centralized execution is generally preferable whenever possible because the monitored hosts bear less of a resource burden. However, remote processing may be unavoidable, or even preferred, in some situations. For large environments with tens of thousands of hosts, centralized execution may be too much for a single monitoring server to handle. In this case, the monitoring system may need to rely on the clients to run their own service checks and report back the results. Some types of checks may be impossible to run from the central server. For example, plugins that check the amount of free memory may require remote execution.

As a third option, several Nagios servers may be combined to form a single distributed monitoring system. Distributed monitoring enables centralized execution in large environments by distributing the monitoring load across several Nagios servers. Distributed monitoring is also good for situations in which the network is geographically disperse, or otherwise inconveniently segmented.

Bandwidth Considerations

Plugins usually generate some IP traffic. Each network device that this traffic must traverse introduces network overhead, as well as a dependency into the system. In Figure 1.1, there is a router between the Nagios Server and Server1. Because Nagios must traverse the router to connect to Server1, Server1 is said to be a child of the router. It is always desirable to do as little layer 3 routing between the monitoring system and its target hosts as possible, especially

where devices such as firewalls and WAN links are concerned. So the location of the monitoring system within the network topology becomes an important implementation detail.

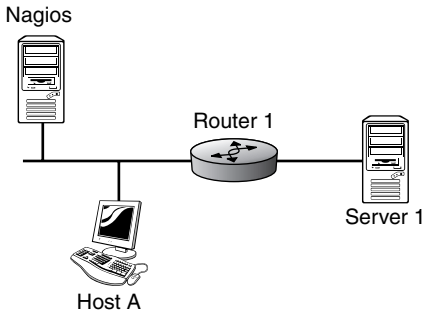

Figure 1.1 The router between Nagios and Server1 introduces a dependency and some network overhead in the form of layer 3 routing decisions.

In addition to minimizing layer 3 routing of traffic from the monitoring host, you also want to make sure that the monitoring host is sending as little traffic as possible. This means paying attention to things such as polling intervals and plugin redundancy. Plugin redundancy is when two or more plugins effectively monitor the same service.

Redundant plugins may not be obvious. They usually take the form of two plugins that measure the same service, but at different depths. Take, for example, an imaginary Web service running on Server1. The monitoring system may initially be set up to connect to port 80 of the Web service to see if it is available. Then some months later, when the Web site running on Server1 has some problems with users being able to authenticate, a plugin may be created that verifies authentication works correctly. All that is actually needed in this example is the second plugin. If it can log in to the Web site, then port 80 is obviously available and the first plugin does nothing but waste resources. Plugin redundancy may not be a problem for smaller sites with less than a thousand or so servers. For large sites, however, eliminating plugin redundancy (or better, ensuring it never occurs in the first place) can greatly reduce the burden on the monitoring system and the network.

Minimizing the overhead incurred on the environment as a whole means maintaining a global perspective on its resources. Hosts connected by slow WAN links that are heavily utilized, or are otherwise sensitive to resource utilization, should be grouped logically. Nagios provides *hostgroups* for this purpose. These allow configuration settings to be optimized to meet the needs of the group. For example, plugins may be set to a higher timeout for the Remote-Office hostgroup, ensuring that network latency doesn't cause a false alarm for hosts on slower networks. Special consideration should be given to the location of the monitoring system to reduce its impact on the network, as well as to minimize its dependency on other devices. Finally, make sure that your configuration changes don't needlessly increase the burden on the systems and network you monitor, as with redundant plugins. The last thing a monitoring system should do is cause problems of its own.

Network Location and Dependencies

The location of the monitoring system within the network topology has wide-ranging architectural ramifications, so you should take some time to consider its placement within your network. Your implementation goals are threefold.

1. Maintain existing security measures.
2. Minimize impact on the network.
3. Minimize the number of dependencies between the monitoring system and the most critical systems.

No single ideal solution exists, so these three goals need to be weighed against each other for each environment. The end result is always a compromise, so it's important to spend some time diagramming out a few different architectures and considering the consequences of each.

The network topology shown in Figure 1.2 is a simple example of a network that should be familiar to any sysadmin. Today, most private networks that provide Internet-facing services have at least three segments: the inside, the outside, and the demilitarized zone (DMZ). In our example network, the greatest number of hosts exists on the inside segment. Most of the critically important hosts (they are important because these are Web servers), however, exist on the DMZ.

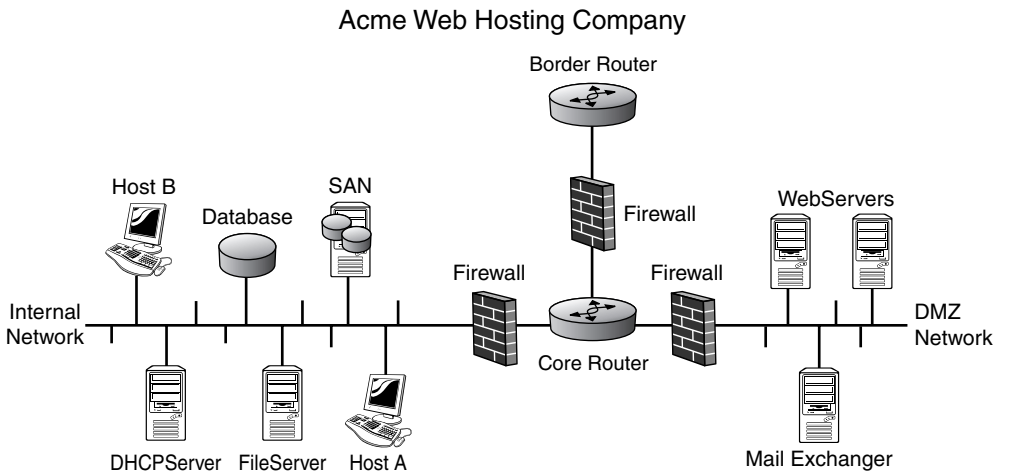

Figure 1.2 A typical two-tiered network.

Following the implementation rules at the beginning of this section, our first priority is to maintain the security of the network. Creating a monitoring framework necessitates that some ports on the firewalls be opened, so that, for example, the monitoring host can connect to port 80 on hosts in other network segments. If the monitoring system were placed in the DMZ, many more ports on the firewalls would need to be opened than if the monitoring system were placed on the inside segment, simply because there are more hosts on the internal segment. For most organizations, placing the monitoring server in the DMZ would be unacceptable for this reason. More information on security is discussed later in this chapter, but for this example, it's simple arithmetic.

There are many ways to reduce the impact of the monitoring system on the network. For example, the use of a modem to send messages via the Public Switched Telephone Network (PSTN) reduces network traffic and removes dependencies. The best way to minimize network impact in this example, however, is by placing the monitoring system on the segment with the largest number of hosts, because this ensures that less traffic must traverse the firewalls and router. This, once again, points to the internal network.

Finally, placing our monitoring system in a separate network segment from most of the critical systems is not ideal, because if one of the network devices becomes unavailable, the monitoring system loses visibility to the hosts behind it. Nagios refers to this as a *network-blocking outage*. The hosts on the DMZ are children of their firewall, and when configured as such, Nagios is aware of the dependency. If the firewall goes down, Nagios does not have to send notifications for all of the hosts behind it (but it can if you want it to), and the status of those hosts will be flagged unknown in availability reports for the amount of time that they were not visible. Every network will have some amount of dependency, so this needs to be considered in the context of the other two goals. In the example, despite the dependency, the inside segment is probably the best place for the monitoring host.

Security

The ease with which large monitoring systems can become large root kits makes it imperative that security is considered sooner, rather than later.

Because monitoring systems usually need remote execution rights to the hosts it monitors, it's easy to introduce backdoors and vulnerabilities into otherwise secure systems. Worse, because they're installed as part of a legitimate system, these vulnerabilities may be overlooked by penetration testers and auditing tools. The first, and most important, thing to look for when building secure monitoring systems is how remote execution is accomplished.

Historically, commercial monitoring tools have included huge monolithic agents, which must be installed on every client to enable even basic functionality. These agents usually include remote shell functionality and proprietary byte code interpreters, which allow the monitoring host carte blanche to execute anything on the client, via its agent. This implementation makes it difficult, at best, to adhere to basic security principles, such as least privilege.

Anyone with control over the monitoring system has complete control over every box it monitors.

Nagios, by comparison, follows the UNIX adage: "Do one thing and do it well." It is really nothing but a task optimized scheduler and notification framework. It doesn't have an intrinsic ability to connect to other computers and contains no agent software at all. These functions exist as separate, single-purpose programs that Nagios must be configured to use. By outsourcing remote execution to external programs, Nagios maintains an off-by-default policy and doesn't attempt to reinvent things like encryption protocols, which are critically important and difficult to implement. With Nagios, it's simple to limit the monitoring server's access to its clients, but poor security practices on the part of admin can still create insecure systems; so in the end, it's up to you.

The monitoring system should have only the access it needs to remotely execute the specific plugins required. Avoid rexec style plugins that take arbitrary strings and execute them on the remote host. Ideally, every remotely executed plugin should be a single-purpose program, which the monitoring system has specific access to execute. Some useful plugins provide lots of functionality in a single binary. NSCLIENT++ for Windows, for example, can query any perfmon counter. These multipurpose plugins are fine, if they limit access to a small subset of query-only functionality.

The communication channel between the remotely executed plugin and the monitoring system should be encrypted. Though it's a common mistake among commercial-monitoring applications, avoid nonstandard, or proprietary, encryption protocols. Encryption protocols are notoriously difficult to implement, let alone create. The popular remote execution plugins for Nagios use the industry-standard OpenSSL library, which is peer reviewed constantly by smart people. Even if none of the information passed is considered sensitive, the implementation should include encrypted channels from the get-go as an enabling step. If the system is implemented well, it will grow fast, and it's far more difficult to add encrypted channels after the fact than it is to include them in the initial build.

Simple Network Management Protocol (SNMP), a mainstay of systems monitoring that is supported on nearly every computing device in existence today, should not be used on public networks, and avoided, if possible, on private ones. For most purposes involving general-purpose workstations and servers, alternatives to SNMP can be found. If SNMP must be used for network equipment, try to use SNMPv3, which includes encryption, and no matter what version you use, be sure it's configured in a read-only capacity and only accepts connections from specific hosts. For whatever reason, sysadmins seem chronically incapable of changing SNMP community string names. This simple implementation flaw accounts for most of SNMP's bad rap. Look for more information on SNMP in Chapter 6, "Watching."

Many organizations have network segments that are physically separated, or otherwise inaccessible, from the rest of the network. In this case, monitoring hosts on the isolated subnet means adding a Network Interface Card (NIC) to the monitoring server and connecting it to the private segment. Isolated network segments are usually isolated for a reason, so at a minimum, the monitoring system should be configured with strict local firewall rules so that they don't forward traffic from one subnet to the other. Consideration should be paid to building separate monitoring systems for nonaccessible networks.

When holes must be opened in the firewall for the monitoring server to check the status of hosts on a different segment, consider using remote execution to minimize the number of ports required. For example, the Nagios Box in Figure 1.3 must monitor the Web server and SMTP daemon on Server1. Instead of opening three ports on the firewall, the same outcome may be reached by running a service checker plugin remotely on Server1 to check that the apache and qmail daemons are running. By opening only one port instead of three, there is less opportunity for abuse by a malicious party.

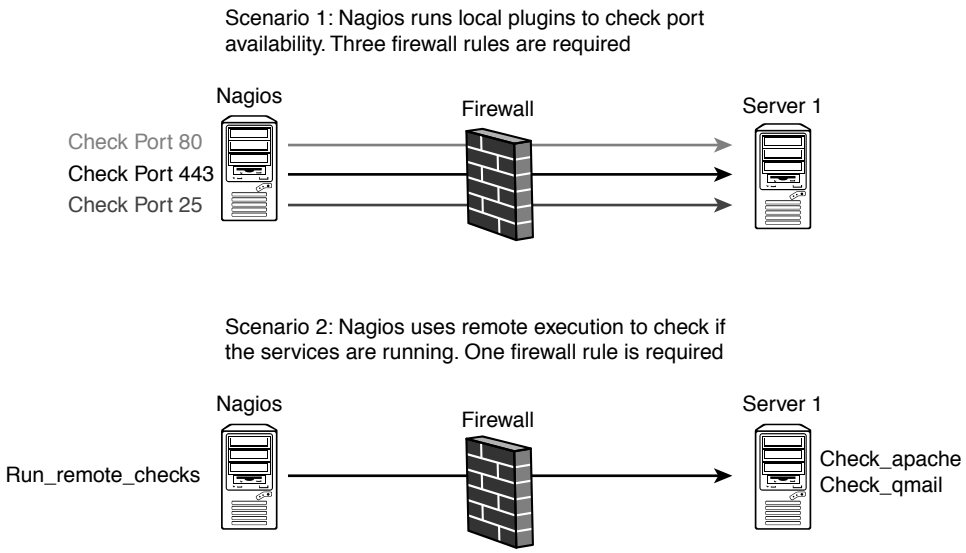

Figure 1.3 When used correctly, remote execution can enhance security by minimizing firewall ACLs.

A good monitoring system does its job without creating flaws for intruders to exploit; Nagios makes it simple to build secure monitoring systems if the implementers are committed to building them that way.

Silence Is Golden

With any monitoring system, a balance must be struck between too much granularity and too little. Technical folks, such as sysadmins, usually err on the side of offering too much. Given 20 services on 5 boxes, many sysadmins monitor everything and get notified on everything, whether the notifications might represent a problem.

For sysadmins, this is not a big deal; they generally develop an organic understanding of their environments, and the notifications serve as an additional point of visibility or as an event correlation aid. For example, a notification from workstation1 that its network traffic is high, combined with a CPU spike on router 12, and abnormal disk usage on Server3, may indicate to a sysadmin that Ted from accounting has come back early from vacation. A

diligent sysadmin might follow up on that hunch to verify that it really is Ted and not a teenager at the University of Hackgrandistan owning Ted's workstation. It happens more often than you'd think. For the non-sysadmin, however, the most accurate phrase to describe these notifications is *false alarm*.

Typically, monitoring systems use static thresholds to determine the state of a service. The CPU on Server1, for example, may have a threshold of 95 percent. When the CPU goes above that, the monitoring system sends notifications or performs an automatic break/fix. One of the biggest mistakes an implementer can make when introducing a monitoring system into an environment is simply not taking the time to find out what the normal operating parameters on the servers are. If Server1 typically has 98 percent CPU utilization from 12 a.m. to 2 a.m. because it does batch processing during these hours, then a false alarm is sent.

False alarms should be methodically hunted down and eradicated. Nothing can undermine the credibility of, and erode the support for, a fledgling monitoring system such as people getting notifications that they think are silly or useless. Before the monitoring system is configured to send notifications, it should be run for a few weeks to collect data on at least the critical hosts to determine what their normal operational parameters are. This data, collectively referred to as a baseline, is the only reasonably responsible way to determine static thresholds for your servers.

That's not to say our sysadmin should be prevented from getting the most out of his cell phone's unlimited data plan. I'm merely suggesting that some filtering be put in place to ensure no one else need share his unfortunate fascination. One great thing about following the procedural approach outlined earlier in this chapter is that it makes it possible to think about the organization's requirements for a particular service on a specific host *before* the thresholds and contacts are configured. If Alice, the DBA, doesn't need to react to high CPU on Server1, then she should not get paged about it.

Nagios provides plenty of functionality to enable sysadmins to be notified of "interesting events" without alerting management or other noninterested parties. With two threshold levels (warning and critical) and a myriad of escalation and polling options, it is relatively simple to get early-and-often style notifications for control freaks, while keeping others abreast of just the problems. It is highly recommended that a layered approach to notification be a design goal of the system from the beginning.

Good monitoring systems tend to be focused, rather than chatty. They may monitor many services for the purpose of historical trending, but they send fewer notifications than one would expect, and when they do, it's to the group of people who want to know. For the intellectually curious, who don't want their pager going off at all hours of the day and night, consider sending summary reports every 24 hours or so. Nagios has some excellent reporting built in.

Watching Ports Versus Watching Applications

In the "Processing and Overhead" section, earlier in the chapter, we briefly discussed redundant plugins that monitored a Web server. One plugin simply connected to port 80 on the Web server, while the other attempted to login to the Web site hosted by the server. The latter plugin is an example of what is increasingly being referred to as End to End (E2E) Monitoring, which makes use of the monitored services in the same way a user might. Instead of monitoring port 25 on a mail server, the E2E approach would be to send an email through the system. Instead of monitoring the processes required for CIFS, an E2E plugin would attempt to mount a shared drive, and so on.

While introducing more overhead individually, E2E plugins can actually lighten the load when used to replace several of their conventional counterparts. A set of plugins that monitors a Web application by checking the Web ports, database services, and application server availability might be replaced by a single plugin that logs into the Web site and makes a query. E2E plugins tend to be "smarter." That is, they catch more problems by virtue of detecting the outcome of an attempted use of a service, rather than watching single points of likely failure. For example, an E2E plugin that parses the content of a Web site can find and alert on a permissions problem, where a simple port watcher cannot.

Sometimes that's a good thing and sometimes it isn't. What E2E gains in rate of detection, it loses in resolution. What I mean by that is, with E2E, you often know that there is a problem but not where the problem actually resides, which can be bad when the problem is actually in a completely unrelated system. For example, an E2E plugin that watches an email system can detect failure and send notifications in the event of a DNS outage, because the mail servers cannot perform MX lookups and, therefore, cannot send mail. This makes E2E plugins susceptible to what some may consider false alarms, so they should be used sparingly.

A problem in some unrelated infrastructure, which affects a system responsible for transferring funds, is something bank management needs to know about, regardless of the root cause. E2E is great at catching failures in unexpected places and can be a real lifesaver when used on systems for which problem detection is absolutely critical.

Adoption of E2E is slow among the commercial monitoring systems, because it's difficult to predict what customers' needs are, which makes it hard to write agent software. On the other hand, Nagios excels at this sort of application-layer monitoring because it makes no assumptions about how you want to monitor stuff, so extending Nagios' functionality is usually trivial. More on plugins and how they work is in Chapter 2, "Theory of Operations."

Who's Watching the Watchers?

If there is a fatal flaw in the concept of systems monitoring, it is the use of untrustworthy systems to watch other untrustworthy systems. If your monitoring system fails, it's important you are at least informed of it. A failover system to pick up where the failed system left off is even better.

The specifics of your network dictate what needs to happen when the monitoring system fails. If you are bound by strict SLAs, then uptime reports are a critical part of your business, and a failover system should be implemented. Often, it's enough to simply know that the monitoring system is down.

Failure-proofing monitoring systems is a messy business. Unless you work at a tier1 ISP, you'll always hit some upstream dependency that you have no control over, if you go high enough into the topology of your network. This does not negate the necessity of a plan.

Small shops should at least have a secondary system, such as a syslog box, or some other piece of infrastructure that can heartbeat the monitoring system and send an alert if things go wrong. Large shops may want to consider global monitoring infrastructure, either provided by a company that sells such solutions or by maintaining a mesh topology of hosted Nagios boxes in geographically dispersed locations.

Nagios makes it easy to mirror state and configuration information across separate boxes. Configuration and state are stored as terse, clear text files by default. Configuration syntax hooks make event mirroring a snap, and Nagios can be configured in distributed monitoring scenarios with multiple Nagios servers. The monitoring system may be the system most in need of monitoring; don't forget to include it in the list of critical systems.