Automatic Discovery of Diverse and Dynamic Network Services

Mikko Pervilä
Department of Computer Science, University of Helsinki
P.O. Box 68, FI-00014 University of Helsinki, Finland
E-mail: pervila@cs.helsinki.fi

Abstract
Common-mode failures have thus far been considerably overlooked in reliability studies. By their definition, common-mode failures correlate heavily, breaking the assumption of probabilistic independence. This article describes how monitoring a live, large-scale network infrastructure provides a realistic environment for analyzing common-mode failures, while also introducing new problems not immediately obvious in analytical or simulation studies. Administrators perform changes to a system whenever new requirements, research projects, or unplanned failures manifest. To distinguish changes from failures, the process from network discovery to service monitoring must be streamlined and enabled to handle extremely diverse installations.

1 Introduction
Common-mode failures (CMFs) or $M$-plex faults have been known to computer scientists since the 19th century, from the days of Babbage and his Calculating Engine [1]. The standard method of tolerating faults is by adding redundancy. Redundancy may be applied in either time, hardware, or software by performing more calculations than required. What makes CMFs so interesting is that by their definition, these faults manifest as failures in more than one repetition. This property breaks many previous failure models.
Despite our knowledge of CMFs, assumptions of independent and identically distributed failures continue to surface in academia. Simultaneously, reports from hardware enthusiasts [2, 3, 4] show an alarming number of faults affecting entire series of products. It seems logical that global processes have not made CMFs any scarcer than in the early days of computation, when computer machinery was uniquely built and maintained by local engineers.

As the methodology and bias of web site reporters is sometimes questionable, an initiative has begun to collect a scientific and open database of hardware failures [5]. A recent line of work by Schroeder et al. [6, 7] explores the field of empirical system reliability [8], within which this article belongs. Studies in the field are categorized by their methodology: measuring real-world systems’ failure properties and occurrence, and then adapting new models to fit the data. These methods are hardly new, but the magnitude and dynamism of the observed systems are.

Current distributed systems and peer-to-peer networks are becoming increasingly heterogeneous or diverse. Diversity has been proposed as a possible cure for software-based CMFs [1]. Instead of repeating the calculations on (almost) identical components, the principle of design diversity advocates introducing heterogeneity to the system on purpose.

To complement previous studies, here the viewpoint is precisely on the diversity of the test subjects. In order to capture the states of a real-world environment, we have built a measurement framework on top of the Nagios sentinal service\(^1\). In this initial phase the goal is to seek the ratio of CMFs to independent failures. In later phases the measurement framework will be continuously adapted to better distinguish possible causes of encountered failures. The hypothesis is that common-mode failures are as common as independent failures. Furthermore, we begin to formulate a conjecture that diversity is beneficial to the fault tolerance of distributed systems, but perhaps harmful for their maintainability.

Common-mode failures and the problem statement are formulated in Section 2. Section 3 describes a framework designed to measure CMFs in a real-world environment, along with the framework’s subsystems under construction. Section 4 describes new difficulties encountered along with possible solutions. Section 5 concludes the contributions of this article.

\(^1\)Available from \url{http://www.nagios.com}
2 Common-mode failures

Fault-tolerance in distributed systems and peer-to-peer networks is normally attained by adding redundancy to the system. Avižienis and Kelly [1] divide redundancy into the three domains of time or repetitions, space or hardware, and program or software. The textbook definition by Tanenbaum and van Steen [10] differs with regards to the domains: their definition divides information, time, and physical redundancies. In both definitions, redundancy can be applied in different measures to multiple domains, meaning that a general $XT/YH/ZS$ fault-tolerant system could perform $X - 1$ redundant calculations in time, on $Y - 1$ hardware components, with $Z - 1$ program codes.

Regardless of what the actual domains are, the definition of common-mode failures remains the same. A single common-mode failure can manifest in multiple redundant repetitions or replicas. This ability can negate the benefits of redundancy altogether, provided that the replicas are similar. Similarity in time manifests naturally as the extra calculations are executed by the same program code. Similarity in software or hardware means that the redundant replicas are either copies of the same software, or hardware devices carrying similar (potential) defects. Schroeder et al. have defined similarity in this context more formally as the autocorrelation [6] of failure events. Roughly, a single component failure with a high autocorrelation factor sharply increases the probability of similar components failing in the near future. This probability then diminishes with time.

Presently, common-mode hardware and software failures are reported by the media as failures of entire model series of hardware or software components [2, 3, 4]. These reports show that autocorrelated failures do occur, but they are still rare enough to warrant attention from the media. On the other hand, “classical” or independent failures have become so well-known that they receive attention only in the most severe of cases. Thus, the research questions that we have set to figure out are as follows:

1. What is the proportion of CMFs to independent failures?
2. What types of CMFs are most common?

The first question stems from the hypothesis that current real-world systems are more vulnerable to CMFs than predicted. The rationale be-
hind this is that a homogeneous environment seems to be easier to maintain than a highly diverse or heterogeneous one. Additionally, subjective factors seem to gravitate real-world systems towards homogeneity. It is common for system administrators to possess intuitive arguments like "machines from vendor A are better than those of vendor B" or "systems from vendor C should be avoided because they are so unreliable".

The second question explores the possibility that CMFs may result more naturally either in software or hardware environments. CMFs in software may be the result of vulnerabilities or flaws in the program code and thereof resulting updates or upgrades. In hardware devices, CMFs may be the result of common manufacturing, shipping, handling, or administration practices.

Ultimately, the purpose of the ongoing research is to prove the conjecture that diversity is beneficial to the fault-tolerance of distributed systems and peer-to-peer networks. In this context, diversity is defined as the intentional introduction of heterogeneity to a given environment. The main drawback is that diversity is notoriously difficult to formally model or express in simulations. Hence, we turn our attention to a real-world environment, and describe its benchmarking in the next section.

3 The Nagios sentinel service

The Nagios measurement framework or is a robust and mature combination of checks to be run against services and the core logic that handles check intervals and interleaving. As a sentinel service, Nagios is able to function as the operative component of self-healing systems [11]. For this work, the initial interest is on Nagios' error-detection and reporting capabilities only. However, all experience gathered thus far seems to support the idea that administrators commonly perform similar investigative and restorative tasks. Assuming proper authentication and authorization, some of these tasks could well be delegated to an automated system.

In contrast to laboratory models, the object we are monitoring is the Computer Science Department’s environment of infrastructure and research services. This means that the set of services is highly heterogeneous, employing both long-lasting and critical services, like e-mail and NFS, but also experimental and transient projects used by a few researchers only.
Along with those network services which are more or less always available, we are also interested in transient services that are online for the duration of a single project only. Conversely to network services, hardware components usually have no monitors to be read remotely. For monitoring components like CPU temperatures, PSU voltages, or fan rotation speeds, we employ a distributed daemon running in the local operating systems. The following subsections describe a new tool designed for the discovery of transient network services and the distribution of checks for services without a network component. Finally, service dependencies are explained as a concept orthogonal to but closely resembling CMFs.

3.1 Mapping with Nmap

The basic idea of using network mapping with Nagios is simple. Any local subnets are scanned through with a well-behaving network discovery tool, e.g., Nmap\(^2\), and the output is then processed into Nagios' configuration files. Any changes occurring in the environment cause Nagios' configuration to slowly become outdated or stale. After the configuration becomes too stale, another scan is performed and the processed configuration files replaced. Currently, we try to scan every three to six months. At the end of each cycle, around 20-50 services have simply disappeared or been reassigned to different hosts.

Automatic discovery of network services and their configuration on Nagios is an old concept, and it has been implemented in multiple addons. Conversely, Nagios' core FAQ list makes a remark that actively discourages automatic discovery of services [12]. The reason why we have both embraced automatic discovery despite the official view and built yet another tool for this purpose is two-fold.

First, along with services, network administrators themselves come and go. In a recent anecdote, a long-term administrator rediscovered a physical server he had set up a year ago for a specific task which had but recently become active. Specially with robust installations, it is not uncommon for a service to become interesting only after it has experienced a (partial) failure. While doing automatic discovery, we have found services which nobody had missed – yet. Our rationale is that it is better to discover too many services and then reduce the monitored set to a

\(^2\)Available from http://nmap.org
reasonable size, than to try and fail to assemble a complete list.

Second, we did perform an extensive search for existing open source utilities built for the matter at hand. Specifically the Nmap2Nagios and Nmap2Nagios-ng projects were very useful at first. Unfortunately, Nmap2Nagios has not been developed in seven years. Nmap2Nagios-ng\textsuperscript{3}, the successor, has been updated more recently, but the program comments were also rewritten in German. For small changes, the language barrier was surmountable, but when more extensive changes became necessary it became easier to just do a complete rewrite.

For the above reasons, we have now released the Nmap3Nagios tool. For more information, please visit the project web page\textsuperscript{4}. Nmap3Nagios is intended to become the first tool in a new toolkit designed primarily to monitor diverse environments. The next utility under development is a visualization aid that transforms Nagios’ archived log files into input presentable by the Timeline widget\textsuperscript{5}.

\subsection*{3.2 Monitoring local devices}

In difference to the already released tool, the implementation of local device monitoring is still undergoing testing. Nagios provides at least three major alternatives for doing local checks. In order of increasing complexity, the alternatives are the check-by-ssh plugin and the addons NRPE and NSCA. For more information, please see the manual entry on distributed monitoring [13]. Trying to anticipate requirements that will appear later on, we have chosen NSCA as the distributed component.

Authorization issues are perhaps the essential difficulty of distributed monitoring. Accessing local devices administration interfaces does require quite low-level privileges on our current operating systems. This means that the distributed component must be carefully given the right privileges and preferrably none but them. These requirements might be conflicting when, for example, monitoring local hard drives S.M.A.R.T. reports is only possible with root-level access to the device, but that access also enables writing to any file system block.

The benefit of using NSCA is that it does not require input from the

\textsuperscript{3}-ng probably stands for "next generation".
\textsuperscript{4}Nmap3Nagios Home: http://www.cs.helsinki.fi/u/pervila/Nmap3Nagios/
\textsuperscript{5}Timeline widget: http://www.simile-widgets.org/timeline/
central Nagios server. Communication flows from the clients to the central server, which makes restricting unwanted access significantly easier. In addition, all communication is encrypted with standard, SSL-based cryptography libraries. Figure 1 visualizes the communication process.

A common enough question concerns why local monitoring performed by an elaborate distributed setup are preferable over the standard solution of using SNMP. The reason for this is that not all devices have SNMP components; using local checks enables the monitoring daemon to run any program code available. This is a considerable advantage, for some server equipment can only be monitored through proprietary software utilities.

### 3.3 Service dependencies

A vaguely related and quite similar concept to CMFs can be found in Nagios’ software and hardware dependencies. Intuitively, it is easy to understand service dependencies through the example of switches. Server environments are usually built in hierarchical star-shaped topologies where multiple host computers are connected to a hardware switch, and the switches are then interconnected through a spanning-tree protocol (STP). If the Nagios monitoring daemon now resides on a host connected to switch A and some services are served by a host connected to switch B,
each service is defined as dependent on both\textsuperscript{6} switch B and switch A. This means that failures encountered in either switch will result in the dependent services’ state being marked as unknown, not as having failed. Simply put, nothing more can then be known about the services, because another failure makes more thorough checking impossible.

Without this feature, it would be quite impossible for Nagios to distinguish CMFs from independent failures causing communication loss to a set of services. The main difference to CMFs is exactly this uncertainty: as the service state can not be known, we have chosen to err on the side of caution and ignore failures possibly masked by dependency failures. As infrastructure malfunctions are luckily quite rare, there remains a possibility of investigating these events case-by-case, perhaps adding discovered causes to the results manually.

With only 19 switches in the department network, our dependency graph is still of a manageable size. Keeping scalability in mind, this is an area where further research will most certainly be necessary. Tools that can form a topology from network addresses do exist, and will be needed if the network size grows. However, there are still more pressing issues in the work queue.

4 Ongoing problems

Hardly any research project finishes without discovering additional problems to be solved. Some of the issues described in this section were accurately predicted at the very beginning of the project. For example, the user interface issues described below have been well-known for quite a while, and multiple addon packages have tried to incorporate enhancements. Others, like the reconfiguration problem of booting multiple operating systems, exist on the boundary of what is feasible to monitor using Nagios. Finally, some problems were simply excluded by limiting the observed set of services. Monitoring the department’s set of laptops was one of these, for even though a distributed monitoring setup using VPN might have been possible, the potential for false positives was judged too high.

\textsuperscript{6}Nota bene: service-to-host dependencies are implicitly handled by the core logic.
4.1 Planning and communication

Perhaps the major problem in the measurement setup stems, unsurprisingly, from the human component. It is easy to understand that any administrator loathes redundancy in her workload, for it is vexing to upkeep the same configuration in different places. There are limits to how much paperwork any worker can be expected to reasonably fulfill. It is possible that with issue tracking, e-mail correspondence, the monitoring setup, and finally, making the actual changes, we are approaching the limit of an administrator’s patience.

We have tried to reduce the amount of configuration tasks by taking the burden of maintaining Nagios from the administrators. This means that one of the researchers has worked part-time for the benefit of the IT team and implemented the monitoring framework on behalf of them. This solution seems to have worked reasonably well. The benefit is that by integrating closely with the IT staff, the measurements are done with a much better background knowledge of IT processes.

On the other hand, the integration is not perfect. There is always a small delay between noticing a failure report and then investigating what the root cause is. In many of the cases, the cause is a failure-preventive task executed by one of the administrators. These effects are difficult to completely classify as either service failures or just glitches with the measurement system. The current solution is to mark all reports consistently as failures, on the basis of the assumption that both independent and CMFs are then similarly measured. Any additional information received from the IT staff is also recorded case-by-case. The final output data will contain both the measurements and background reports investigated.

As a more long-lasting solution, we are constantly evaluating Nagios user interface addons. The idea in this is that a more tempting GUI would make it easier to receive input directly from the administrators.

4.2 User interface issues

Not all service failures do ever get repaired. For some events, the fix may be performed by moving the service to another host, merging the service with another existing service, creating a new service, or simply removing the old service altogether. In these cases, the failure is usually acknowledged and/or further notifications surpressed. If manual pruning
is not regularly performed, the number of these obsolete services can make distinguishing new failures difficult.

Unfortunately, the core Nagios web GUI does not contain any reconfiguration features. An extensive number of GUI addons have been released, and we have evaluated almost every one of them\(^7\). For a partially complete listing, see [4]. With the exception of the Centreon\(^8\) project, most of the addons have a very similar binding problem. Typically, each addon incorporates the core Nagios’ reporting features and adds generative configuration facilities. This means that a new GUI can be used to define templates and individual services, and once their configuration is complete, the entire configuration is output and Nagios’ core daemon restarted. The binding problem revolves around this duality of reporting and configuration, as the reporting interface has been left as is. This means that when a failure has been reported, an administrator can view the problem report from the reporting interface, but must look for and reconfigure the service from a separate interface. This flaw seems trivial to begin with, but causes a severe hindrance in practice.

Very recently, two projects with novel approaches for the GUI have been announced. One of them is the NINJA project\(^9\) written by a group of dedicated software developers. Another is the upcoming Nagios XI\(^10\) release offered by a team closely related to Nagios’ core. As of writing, these two are the projects that we have least experience with, but are working on testing both as soon as possible. The main hindrance to evaluating GUI addons is their configuration complexity. Centreon is a prime example of this, as the software is primarily sold as a support service.

### 4.3 Delicate instrument

For the analysis of results, Nagios is a demanding system. The reason for this is that Nagios’ check plugins can be very delicate. Even with reasonable definitions of failures, e.g., at least five-minute downtimes, Nagios can collect a surprising amount of data. Failures normally invisible to both users and administrators surface regularly, specially those that

\(^7\)As of writing, many of the open source projects have long since been abandoned.
\(^8\)Centreon: [http://en.doc.centreon.com/Main_Page](http://en.doc.centreon.com/Main_Page)
\(^9\)NINJA: [http://www.op5.org/community/projects/ninja](http://www.op5.org/community/projects/ninja)
occur during the hours of least usage. For example, backup schedules and automated software upgrades may give cause for a service to be taken offline for a few minutes in the small hours of the night.

Typically, users are blissfully ignorant of these service windows, while Nagios remains ever vigilant. The downtime is a problem only in the sense that failure severity depends only indirectly on daytime, while Nagios approach is somewhat based on it. Also, some of our researchers and students keep working hours which might be classified as odd. Therefore, we have eschewed silencing failure notifications based on classical working hours, at the cost of suffering a somewhat more chatty monitoring setup.

4.4 Reconfiguration

Currently, the most difficult problem to solve remains the automatical reconfiguration of monitored services at run-time. More specifically, the problem occurs in at least two very different cases. It is encountered in workstations with multiple operating systems (OSs), and it is also encountered with network devices able to reconfigure pathways when necessary.

For multiple OSs, reconfiguration must be done when a different OS is rebooted. Each of the department’s workstations runs a local distribution of Linux as the default operating system. Additionally, a significant number of workstations is also installed with Microsoft Windows XP. In our classroom configuration, any student is able to change the OS at the start of her session. This freedom of choice requires the distributed component of Nagios to be configured twice: once for each OS. Our prototype versions of NSCA are based on the default Linux configuration. For Windows, the system is somewhat more complex as the NSCA component is not exactly the same program. We are trying to make the change transparent to the central Nagios service, so that reconfiguration would be necessary only at the client end. This would enable the central Nagios to reuse the same service and host names, bypassing reloads of the central configuration.

Using a spanning-tree protocol (STP) for network switches is more or less the de facto standard for any larger ethernet segment. STP provides a major benefit as the switches are able to renegotiate alternative pathways should an interconnected link fail. But STP also causes difficulties for any services depending on the switch pathways, i.e., all services not on the same host as the central Nagios itself.
As of writing, no solution for STP reconfiguration has been found. In theory, the reconfiguration step should be based on an informed topology change, which would seem to require getting information from the switches. This means that configuring the initial network dependencies and STP reconfiguration should optimally be processed by the same utility. The proprietary nature of the hardware devices make this less than trivial, but the problem is somewhat mitigated by its rarity. Currently, the network dependencies are based on the default interconnections, and changes to the topology are resolved manually.

5 Conclusion

Despite the number of problems encountered, and that this article describes a measurement project in its early phases of implementation, the research questions presented seem feasible to answer. The failure monitoring service that we have devised is clearly not a fire-and-forget solution, but one that requires constant nurturing. This effect seems to be in line with the predicted difficulty of administering truly diverse environments, however, and not a fault of the Nagios sentinel service.

At this point, common-mode failures seem very much real and not just an academic exercise. Later publications will present measurement results attained, along with the actual data for further analysis. Even though many of the details are in flux, and not many conclusions have yet been made, research in this field seems both inspiring and necessary for the well-being of computer systems and networks to come.

Acknowledgments

The IT staff of both the Department of Computer Science and the Helsinki Institute of Information Technology (HIIT) for their patience.

References


