

Using Nagios to monitor faults in a self-healing environment

Mikko A.T. Pervilä

Abstract—The concepts of detecting and monitoring faults in a self-healing environment are examined. The Nagios fault-detection system is evaluated and its self-healing capabilities researched. This paper outlines possibilities of interconnecting Nagios with other applications in order to further facilitate and automate recovery after service failures.

Index Terms—Dependability, fault-tolerance, Nagios, self-healing.

I. INTRODUCTION

DETECTING failures is never easy. It is, in fact, so hard that many of the problems studied under computability theory build on the fact that a Turing machine cannot know when another Turing machine has halted. If we accept the idea that a Turing machine is an accurate abstraction of the modern computer, we must also accept that some faults will remain invisible to the machines.

Yet we humans somehow seem to be able to instantly recognise when a computer is not functioning correctly. True, we might not always be accurate either. Many times a perceived fault should not be blamed on the program, which is only executing as specified. The fault that we detected does not lie within the computing devices, but somewhere else.

But we feel that we are, at least, able to tell when a computer has halted and not doing things like we want it to. The concept of time seems to intuitively be linked with the idea that a program “is taking too long” to process.

Thus, when developing computer programs that monitor other programs for failures, the logical conclusion is to include timers. The monitored program might be allowed leeway in processing, until the inevitable alarm of the timer. The developer is content and moves on to the next problem.

Unfortunately, not all faults are caused by the targeted program. The new problem: how to distinguish the source of the fault, and if the source is not of concern to us (not our problem), how to ignore it. The concept of retry is added. Now the monitoring program tries, times out and retries, hoping that the fault-that-is-not-ours goes away. Many times this technique is sufficient and faults are eliminated. We label those faults as transient: they come and go, but we can live with them.

Checking and retrying quickly becomes resource intensive. We do not want the monitoring program to perform the tests again and again. Better to let it rest between executions – after all, we can wait a little before detecting a new problem.

If the monitoring is not constant, as often is the case, some part of the faults that the system encounters may not be detected. In distributed systems, transient faults become more common as the systems encompass a wider area of the environment. Because of this, care has been taken into building communication channels that are dependable even in the presence of faults. With the use of standard technologies like TCP, we are more interested in catching only those failures that require intervention from the administrator.

With more complex systems the administrator becomes easily overworked and stressed. The monitoring programs discover too many faults, and not all of them are important enough to warrant immediate attention. The administrator is human – she makes mistakes [1]. Thus we must now turn our attention into diminishing the amount of work it takes to repair such failures. After detection and monitoring, automating the repairs is the next logical goal on the track towards self-healing systems.

In this article we examine how the Nagios monitoring application is able to provide solutions for the self-healing problem space. Nagios predates many of the concepts of self-healing and builds on fault-tolerant concepts and techniques. Special interest is taken into how Nagios can be extended from a fault-detection application into a part of an autonomous system.

The possibility of interconnecting Nagios with other systems might provide ideas for further studies. Koopman's article [2] is used as a basis for the concepts and terminology relating to self-healing. The model for faults, errors and failures is from Avižienis *et al.* [3].

II. DETECTION AND MONITORING

One of the most common techniques for detecting faults in distributed environments is observing the state of services with a dedicated application. This application can be called a *sentinel service* [2], referring to its nature as a watchdog for failures in system operation.

The detection of service faults is often based on probabilistic decisions concerning the type of service and a time frame in which it should respond to a query. Such decisions are made by gathering and processing information relating to the relevant services. A singular, discrete decision is called a *check*. Performing a check may be a simple decision based on the existence of a process within the operating system's memory. It can also be a time-consuming query for multiple data sets, followed by a heuristic decision of service correctness.

Manuscript received March 6, 2007.

Mikko Pervilä is with the Department of Computer Science, Helsinki University, 00014 Helsingin yliopisto FINLAND (corresponding author to provide phone: +358-44-7696086; e-mail: pervila@cs.helsinki.fi).

Since nondeterministic changes in the environment can cause both transient and permanent faults to appear, the service checks will have to be repeated at specified intervals. The performance of repeating checks over a time period is called the *monitoring* of a set of services. Monitoring a service may include multiple different check types that are performed at both regular intervals and after specific changes in the environment.

A. Detection techniques

The detection techniques employed in monitoring can be categorized into two sets according to the visibility of the checks to the service being monitored [2].

Nonintrusive methods try to predict the state of the service by monitoring its external attributes. These might be the number of system calls that the process makes, its memory or CPU time usage or the messages it sends to the environment. The service can be aware of the monitoring sentinel, sending specially crafted status messages to it. Example targets for nonintrusive checks are SNMP, system loads and the temperature and humidity within a computing site.

Intrusive methods concentrate on acceptable input/output combinations on the service. Their application can be seen to change the service state, albeit this change can be a transient one. Intrusive sentinels can send queries to the service, either to the same interfaces that are used to serve users or to interfaces that have been developed for testing purposes. Examples of intrusive checks are queries for specific pages from a HTTP server, inserting and deleting a row in a database and injecting faults in order to test fault-tolerant systems.

A sentinel service may employ a combination of both intrusive and non-intrusive methods in order to monitor one or several services. Special program logic must be present to handle situations where the different testing methods yield conflicting results for a specific service. The sentinel may default to non-intrusive methods but change to an intrusive method when a possible failure is detected.

Upon detecting a failure, steps must be taken into correcting the problem. Currently the most used approach is to inform the system administrators and let them handle the repairs. With the progress of self-healing techniques, the alternative of letting the sentinel do a larger part of the work becomes more tempting.

Preliminary tasks could include collecting information about the failure before notifying the operators. Further operations should be done carefully, so that the automated repairs will not cause any more damage to the system. For example, rebooting a host just because one of its services has failed is clearly not desirable if the other services on the host are still operating correctly.

III. THE NAGIOS MONITORING APPLICATION

Nagios is a sentinel service that concentrates on monitoring failures and reporting their existence to selected destinations. Nagios has been designed for the Linux operating system, but it is possible to install Nagios on most other UNIX variants. The sentinel has been developed as an open source project and released under the GNU General Public License version 2.

The development started in 1999 and continues to present day – as of writing, the current version is 2.7. It is considered quite mature for use in production environments. Several books have been written about the application [4].

It is worth noting that the development and distribution of the application have benefited from splitting the project into different parts [5]. The main application is responsible for scheduling and executing the service checks in a concurrent fashion. It also maintains all state information and takes action when the transitions require it. The checks are called *Nagios plugins* and are available separately. Their use is recommended but not necessary – the administrator may set up a system that only executes locally developed service checks. Additionally, the core addons include the NRPE (Nagios Remote Plugin Executor) and NSCA (Netsaint/Nagios Service Check Acceptor) tools. NRPE is used to execute indirect checks locally on target hosts. NSCA enables *passive monitoring*, where Nagios is split into client and server applications. This technique is useful in physically distributed setups and it is closely related with the scalability examined in section IV.

A. Specifying a check and running it

In Nagios, the size of the self-healing unit is a service. The services are monitored by the main application running a set of checks against the specified services. Each service is announced using the hierarchical, template-based configuration file system of Nagios. The configuration files are human-readable text files that follow the syntax defined in Nagios' documentation [6]. The documentation is considered to be very complete; only a rough outline of the configuration process is given here. Because of the size of the documentation, additional references are made when necessary.

The configuration files can be split into separate directories and processed recursively. Depending on the amount of services, this may simplify generating and grouping the configuration files.

The definitions in the files specify Nagios *objects* and their interrelations. Using templates and inheritance as a basis for definitions is strongly encouraged. Examples of objects are hosts, service dependencies, services and commands.

Hosts serve services. Each service is defined to belong to a specific host². When all services on the host are working correctly, the host is taken to be working correctly as well. After a service on the host fails, the host status can be checked separately. If the host is considered to have failed as well, service checks are still processed but the resulting notifications are silenced. Notifications are enabled when the host resumes operation [8]. As we will see later on, recovery for multiple services begins with their host.

Service dependencies define relations between two services that are being monitored. The logic is simple: if a service fails, the depending service will fail as well. Using dependencies, the monitoring system can skip redundant checks and notifications. Some of the graphical addons available to Nagios are able to draw dependency graphs for services.

¹ Netsaint is an obsolete name of the Nagios project.

² Special care has to be taken when defining clustered services [7].

Services are defined very loosely to be whatever that can be monitored. The definition includes several obligatory fields, for example the check interval, maximum retries and a check command. Intervals and retries are used to deduce the state of the service in the presence of transient faults. The values are defined in time units; flexible setups are possible, and Nagios tries to follow all the definitions. When monitoring larger setups, this may require extending the resource usage of Nagios through additional concurrent checks. The target of the check and additional information can be defined through the use of macros and command line arguments. They are substituted and passed on to the check command, which is an executable plugin.

B. Plugging in a check

The plugin package contains instructions for extending the set of plugins with locally developed ones as needed. In the open source fashion, new plugins are readily accepted for community development and eventual inclusion in the plugin package. The current version (1.4.6) of the package includes over 100 plugins as small, executable files. About half of the executables are written in Perl, a bit less than a half in C and the remaining as shell scripts.

Following the idea of architecture completeness, the main application does not need to know how the plugins are implemented. The standards that acceptable plugins *should* conform to are explicitly written out in the Nagios developer guidelines [9]. In practice, the administrators have total freedom with their own systems. The full guidelines are beyond the scope of this article, but the basic input / output requirements are examined.

Input. The plugin should read two argument values as the warning and critical ranges for the check in question. Further arguments may be given in the command specification for the check in question. The plugin should provide documentation for these in the form of help messages. The argument values are used to modify check behavior. Depending on the values and the parsed output of the monitored service, the plugin will deduce the service state and report it to the main application.

Output. Every Nagios plugin may print a single line of output containing the relevant results for the current check. The length of the output line should not exceed 80 characters in order to guarantee successful notification on mobile phones, pagers and other devices where the display size is small. The state of the service is passed to the main application as a return code – this is the only real requirement from a functional plugin.

C. Changing the service state

The basic state labels [10] for a service or a host are “OK” or “up”, “warning”, “critical” or “down”, “unknown”, and “unreachable”. The first two states signal correct functioning and the rest are error states. In addition to the state labels, Nagios defines a *state type* on the basis of how many times a check has indicated that a transition should be made.

The object definition files define a parameter for maximum check attempts³. When the number of checks done is less than the value of the parameter, the transition is considered *soft*.

When the number reaches the value, a *hard* transition is made. A recovery from a soft state is considered soft; respectively a recovery from a hard state is a hard recovery. A change from a soft state to a hard state causes a *notification*, which we will explore more fully in section IV.

There are a number of exceptions to this logic caused by the flexibility of the object definitions. Due to the variable parameters and the combination of a state type with the states themselves, drawing a state diagram becomes quickly counterintuitive. We will not try to catch all the exceptions or possibilities, but show in what way soft and hard transitions are beneficial for self-healing purposes. But before that, we must define when a transition is *not* made.

D. Cascading failures

Nagios provides support for network topologies in multiple ways. Here, the notion of system self-knowledge is tied with the concept of cascading failures. Whenever a service or host fails, the state of the depending services cannot be known. The services might be functioning correctly, but other failures prevent the sentinel from reaching the services. The basic types of cascading failures are host, parent and dependency disruptions.

The *host status* is deduced from the state of the services running on it. If even one of the services is still available, the host is considered to be available as well. However, in some cases a host may have failed even when its services are still available. For example, DNS checks defined for a host may fail independently from the host. In situations related to cascading failures, a host's status may be rechecked even while its services are still marked as available. This is commonly done to figure out the root cause of a problem affecting multiple related hosts or services.

Parent relations are specified between hosts. They are routinely used in defining routers and firewalls that might disrupt network communications upon failure. If a remote service check returns an error state, Nagios will walk the parent tree until it reaches the root or a functioning parent. After this, the malfunctioning child is marked as “down”, and the rest of the hosts below it as “unreachable”. Notifications for unreachable hosts may be suppressed with the object definition files. A CGI script is provided to view network outages [11], several addons for further visualization exist [12].

Dependencies between hosts and services are defined separately in the definition files [13]. The dependencies are not inherited by default⁴ and their use is considered an advanced feature. With dependencies, relations can be specified between services on different hosts. Dependency definitions allow total control of check execution and notification suppression. E.g. a local application server might be dependent on a VPN tunnel to a central database server, although both may suffer failures independently.

Since the number of dependencies can grow quickly, it should be noted that a suitable shorthand exists for writing multiple objects with a single definition [14].

In addition to cascading failures, checks and notifications can be deliberately turned off with scheduled downtime or

³ *max_check_attempts* in both service and host definitions

⁴ Although, in the same flexible fashion, they can be made to inherit.

when a service is detected to have begun *flapping*. Flapping is used to distinguish situations where the objects change state too frequently, causing too many notifications. Flapping [15] remains an experimental feature and has to be separately enabled. It is designed to be useful in situations where a service or host state changes too frequently, causing an unwanted number of notifications.

IV. REPORTING AND REPAIRING PROBLEMS

The concepts of *event handlers* and notifications in Nagios raise in importance when discussing self-healing implementations. Event handlers are run in all state changes and they allow both proactive fault tolerance and retroactive reconfiguration steps. Notifications are primarily sent out when a service or a host does a hard state change.

A. Reconfiguration steps

Event handlers are defined in the same object definition files as any other Nagios object [16]. Their use resembles plugins in that each event handler is defined as a command appended with possible arguments and environmental values. The use of event handlers is left to the administrators, though examples, ideas and community-supported instructions are available. Following GNU/Linux ideas, an event handler can be *any* executable program. The minimum requirement for an event handler is that it should read the event type that caused the execution of the handler and react accordingly. The event type is passed as the first command line argument.

A very basic type of an event handler might do a remote login to the target host and restart the failed service. This proactive repair would take place when the soft state change takes place. Event handlers are only run once for each state change, i.e. the service restart would be done upon the initial discovery of a service failure, but not on subsequent retries. If the restart is not effective in repairing the failure, the event handler will eventually be called for a hard state change. (Note that the current state will still be marked as soft, but the check attempt will match the maximum retry count.) This time the handler might create a problem ticket into a local job database and append it with the information gathered so far. When the maximum retries are reached, an eventual notification will be sent to service contacts. The personnel could check the ticket and proceed with manual repairs, perhaps later on improving the soft event handling with additional intelligence.

A. Notifying users and administrators

When a notification has to be made, the definition files will be again consulted to find out who will receive the information and how the transfer will proceed. Contact groups and schedules are specified in the host and service definitions. The contact groups may overlap. Schedules can be used to limit the reception of some notifications to working hours. E.g. in hosting environments only a subset of the personnel is scheduled to be on-duty and may be notified about failures day or night.

The options for notifying contacts are diverse, and it may be beneficial to use several methods at the same time. Email is probably the most commonly used method, but since the

notification is done by calling an executable file, any method that can be programmed is suitable. It is useful to extend Nagios with SMS services, or pagers in other areas. Instant messengers and VoIP calls might be suitable alternatives as well.

All of the contacts need not be humans. Piping the notification messages to a separate parser or a long-term database might be a better alternative than event handlers for some environments.

B. Long-term information

Since Nagios can be configured to log all of the state changes, it is usable for monitoring service level agreements and overall trends. The statistical functions of the main application and its CGI scripts can be further enhanced by the event handlers. A global event handler could be defined so that all information gathered by Nagios is also forwarded to a separate database. Thus, more advanced data mining operations can be performed on the database.

The built-in displays are probably sufficient for most cases. Different user levels can be defined with separate visibility levels. For example, a customer contact could be given permission to view, but not modify, the service states pertaining to the customer's hosts. Both long-term statistics and current events would then be available to the contact. If the contact is a trusted one, its capabilities could be extended by allowing commands to the main application be given through its web interface.

When failures occur, Nagios supports repair communications through its web interface. The personnel working on the problem may write out additional details and estimated repair times. All are stored for latter viewing and analysis, so that the repair process may be timed, examined and improved. Troublesome services that are the cause for multiple failures may be replaced or eliminated.

V. SCALING AND INTERCONNECTING NAGIOS

Nagios enjoys a large user base and is considered to be a reliable sentinel service for a wide scale of services. The current web site includes an optional feature for adding user profiles. It is meant as a voluntary method of disclosing statistical information [17] about Nagios setups. As of writing, the largest implementations constitute of over 5 000 hosts or over 30 000 services. It is clear that networks of this size require some special considerations concerning in which manner the checks are scheduled and distributed.

A. Distribution and scaling of a setup

Whenever scaling a Nagios setup for larger environments, some problems are expected and must be dealt with. The most common scenarios involve scheduling problems caused by the amount of checks and communication problems due to different subnets, firewalls and other communication gateways.

Scheduling problems may surface when the amount of services increases. The amount of resources required depends on the type of checks to be run. When the product of the amount of checks and execution time per check exceeds the checking interval, some of the checks will be delayed. This

will manifest in checks running less often than specified and can be verified from the execution logs. The problem may be alleviated by increasing the resources available to Nagios. In particular, the amount of allowed concurrent processes is paramount. Also, even small performance improvements in plugin execution may yield significant increases in overall time taken. Plugins written in C execute typically faster than the interpreted ones.

Communication problems are encountered when some of the services are situated behind firewalls or low-bandwidth links. They are not only caused by the upwards scaling of the Nagios environment, but may be encountered even in very small setups. The Nagios addons NRPE (Nagios Remote Plugin Executor) and NSCA (Netsaint/Nagios Service Check Acceptor) offer different ways of solving problems caused by firewalls or plugin types. The downside is the amount of complexity, specially when NSCA is concerned.

NRPE is used to run plugins indirectly on local hosts [18]. The primary use is to check those services that do not include network communications. Examples are CPU loads, memory usage and the status of RAID arrays. These checks are typically nonintrusive. NRPE has a plugin and a daemon to be run on the target host. In situations where communications are limited by a network boundary, the NRPE daemon may be further delegated to check other services within the boundary. The downside is the vulnerability of the NRPE daemon. If it cannot be reached, all service checks delegated to it are unavailable as well.

NSCA is used to distribute monitoring [19] to other sites within the monitoring environment. The others sites will run secondary instances of the Nagios application, complete with their own set of plugins. The secondary instances will deliver their status information using the NSCA client program. The clients communicate with the NSCA daemon on the central server. The added benefit of this setup is that some of the checks may be delegated to the secondary servers. When specified thus, the sites may even overlap. Notifications are typically triggered by the *staleness* of the information delivered. That is, if no new updates are received within a specified time frame, communications are assumed to be lost and notifications are triggered.

B. Making it redundant

Delegating the responsibility of proactive repairs to Nagios without considering the possibility of the sentinel itself failing would not be a long-lived solution. There are two methods for creating fault-tolerant Nagios applications [20] on the network. One involves event handlers and the other relies on cron and NRPE.

Redundant monitoring involves having two instances of the Nagios application executing service checks concurrently. The downside of this setup is the added overhead of running the checks twice. Although most of the checks are quite conservative in their use of bandwidth, access statistics on the services themselves may become unintentionally skewed if the checks are intrusive. Naturally, this will happen with a singular installation as well, so the problem need not be major. The secondary Nagios server works as a standby spare: its notifications are disabled, but it also checks the availability of

the primary Nagios server. If the check returns an error state, external commands are used to enable notifications and thus assume primary status. If the primary server recovers, notifications are again disabled. A side effect of the recovery is that it may cause blackouts during which neither server monitors the network. The duration of the blackouts depends on the scheduling intervals.

Failover monitoring differs in the activity of the slave server. In this setup, active checks are disabled as well as notifications. The slave host monitors the primary server's status locally with the help of the NRPE addon. If the check indicates an error state, external commands are again used to enable service checks and notifications.

In principle, there can be several servers doing failover, as long as the NRPE checks do not interfere with each other. The downside is that if the primary server is down for extensive periods, the event databases will become inconsistent. This problem may be solved using global event handlers and an external database, as outlined in section IV.

C. Compatible applications

Due to the open source community, Nagios has been extended with many other addons and extensions. This paper examines a very select few of them in order to provide ideas for further research. The selection has been assembled with self-healing attributes in mind.

Perhaps one of the most interesting extensions for the monitoring setup is the possibility of connecting external sensors [21]. With the help of sensors, Nagios may monitor environmental attributes like humidity, temperature, the status of UPS devices and physical intrusion. The event handlers can be configured to manage power events as a last resort method.

Nagios has been successfully interconnected [22] with Cfengine [23], an autonomous agent for system and network configuration. In this setup, Nagios uses event handlers to execute Cfengine's problem-solving functionality whenever a service failure is detected.

Through the use of volatile services [24], Nagios can receive events that are not indications of a state change. These events may be SNMP traps or information about blocked intrusion attempts at a local firewall. Suitable follow-ups include notifying contacts or executing special event handler. SNMP monitoring is also useful for devices where the NRPE plugin cannot be executed, for example routers and switches with a limited operating system. Note that ports exist for Windows NT/2000 platforms [25, 26].

VI. ACKNOWLEDGMENT

Ethan Galstad, the current project leader of Nagios, deserves many thanks for his corrections and clarifications on the technical details presented herein. The DNS example from section III is taken from correspondence with him.

VII. REFERENCES

- [1] A. Brown, D.A. Patterson, "To err is human". *Proc. of the first workshop on evaluating and architecting system dependability, Gothenburg, Sweden, July 2001.* <http://roc.cs.berkeley.edu/papers/easy01.pdf>. [6.3.2007]

- [2] P. Koopman, "Elements of the self-healing system problem space". *Proc. workshop on architecting dependable systems*, 2003
- [3] A. Avižienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure Computing". *IEEE Trans. on dependable and secure computing* Vol. 1, Issue 1, pp.11-33, 2004.
- [4] Nagios: Books. <http://www.nagios.org/propaganda/books/>. [6.3.2007]
- [5] Nagios: Downloads. <http://www.nagios.org/download/>. [6.3.2007]
- [6] Nagios documentation: "Table of contents". http://nagios.sourceforge.net/docs/2_0/toc.html. [6.3.2007]
- [7] Nagios documentation: "Monitoring service and host clusters". http://nagios.sourceforge.net/docs/2_0/clusters.html. [6.2.2007]
- [8] Nagios documentation: "Determining status and reachability of network hosts". http://nagios.sourceforge.net/docs/2_0/networkreachability.html. [6.3.2007]
- [9] Nagios Plugins Development Team, "Nagios plug-in development guidelines". <http://nagiosplug.sourceforge.net/developer-guidelines.html>. [6.3.2007]
- [10] Nagios documentation: "State types". http://nagios.sourceforge.net/docs/2_0/statetypes.html. [6.3.2007]
- [11] Nagios documentation: "Network outages". http://nagios.sourceforge.net/docs/2_0/networkoutages.html. [6.3.2007]
- [12] NagiosExchange: Categories: AddOn Projects: Charts. <http://www.nagiosexchange.org/Charts.42.0.html>. [6.3.2007]
- [13] Nagios documentation: "Host and service dependencies". http://nagios.sourceforge.net/docs/2_0/dependencies.html. [6.3.2007]
- [14] Nagios documentation: "Time-saving tricks for object definitions or how to preserve your sanity". http://nagios.sourceforge.net/docs/2_0/templatetricks.html. [6.3.2007]
- [15] Nagios documentation: "Detection and handling of state flapping". http://nagios.sourceforge.net/docs/2_0/flapping.html. [6.3.2007]
- [16] Nagios documentation: "Event handlers". http://nagios.sourceforge.net/docs/2_0/eventhandlers.html. [6.3.2007]
- [17] Nagios: "User profile stats". <http://www.nagios.org/userprofiles/quickstats.php>. [6.3.2007]
- [18] Nagios documentation: "Indirect host and service checks". http://nagios.sourceforge.net/docs/2_0/indirectchecks.html. [6.3.2007]
- [19] Nagios documentation: "Distributed monitoring". http://nagios.sourceforge.net/docs/2_0/distributed.html. [6.3.2007]
- [20] Nagios documentation: "Redundant and failover monitoring". http://nagios.sourceforge.net/docs/2_0/redundancy.html. [6.3.2007]
- [21] Nagios: "Automation and environmental monitoring products". <http://www.nagios.org/products/>. [6.3.2007]
- [22] G. Retkowski, Building a self-healing network. *O'Reilly ONLamp.com*, 25.5.2006, <http://www.onlamp.com/pub/a/onlamp/2006/05/25/self-healing-networks.html>. [6.3.2007]
- [23] "Cfengine – an adaptive system configuration management engine". <http://www.cfengine.org/>. [6.3.2007]
- [24] Nagios documentation: "Volatile services". http://nagios.sourceforge.net/docs/2_0/volatileservices.html. [6.3.2007]
- [25] Nagios FAQ: "How can I monitor Windows NT / 2000 Servers?". http://www.nagios.org/faqs/viewfaq.php?faq_id=32. [6.3.2007]
- [26] "NSClient official site", <http://nsclient.ready2run.nl/>. [6.3.2007]