SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(14), 1519–1530 (10 December 1998)

Automated System Administration with Feedback Regulation

MARK BURGESS
Centre of Science and Technology, Faculty of Engineering, Oslo College 0254, Oslo, Norway
(e-mail: mark@iu.hioslo.no)

SUMMARY

The automation of system administration tasks requires a notion of convergence towards a stable state. Some simple models for such convergence with feedback, utilizing the system administration robot cfengine, are examined. Statistical analysis of computer systems can provide information which may be used to regulate the way in which they are used in the future without the need for excessive human intervention. © 1998 John Wiley & Sons, Ltd.

KEY WORDS: feedback regulation automatic system administration

INTRODUCTION

The level of human involvement in the running of computer systems is unacceptably high. Errors and maintenance tasks involve human intervention at every level of operation. In many cases, the best model for system maintenance is to wait until the system fails and then try to fix it: more often than not, to reboot. Biological and social systems of comparable and greater complexity have self-healing processes which are cruical to their survival. It will be necessary to mimick such systems if our future computer systems are to survive in the complex and hostile environment of the net.

In essence, what computer systems need is an immune system. Although this inevitably makes one think of computer viruses, there is much more to the maintenance of the body than fighting mischievous programs. An immune system is involved in preventative work and the disposal of waste products, and is essential for the continuous operation of complex social and biological systems. This idea has already been explored in detail[1] and will not be reiterated here. The purpose of the present paper is to examine the present state-of-the-art in automatic feedback maintenance for Unix-like systems.

CURRENT SOFTWARE SYSTEMS

Present day computer systems were not designed with immunity in mind, nevertheless most of them are flexible enough to be able to admit significant new systems. How

CCC 0038–0644/98/141519–12$17.50

Received 15 April 1998
Revised and accepted 14 July 1998

executed too often, essentially 'spamming' the system.[9] This is not unlike the mechanism in the brain whereby neurons have a certain dead-time, except that the dead time is configurable in cfengine.

Actions fall into classes. Cfengine begins by detecting its run time environment: the type of operating system, membership to abstract groups and the date and time of day. This information is summarized in the form of classes, which are then used to label actions in the program.[6,7] Classes can also be defined based on user specified criterea. In this way, changes in the state of the system lead to customization of the executed program. The adaptablity of a cfengine configuration depends therefore on the power of its classing engine. Originally, this was limited to the information which was easily obtainable at the instant at which the program was started. However, this proves to be inadequate for implementing the kind of associative actions required in an adaptive system.

The reactor design of cfengine has proven basically sound, but several additions to the way in which classes could be defined were added in order to allow it to competely replace human handwork in the largest number of situations. The manual definition of classes was introduced to allow switching from the command line or from within a script. This allows parts of the generic cfengine program to be invoked by an **exec()** in response to 'cries for help' by other systems, in response to special situations with a customized response for each occasion. One possibility which springs to mind is network monitoring programs such as Bro[10] and SWATCH[11] which look for specific occurrances of log messages or network activity which resemble system cracking. Rather than have these early warning programs respond by sending warnings, they could be made to call up cfengine with a command line of the form

```
cfengine -Dspecial_signal
```

leading to the inclusion of a special sub-program to deal with the contingency:

```
import:
 special_signal::   cf.special
 swatch::           cf.swatch
```

With manual class activation, cfengine may be called up by any root process in order to come to the system's rescue. Over-execution from calls by multiple processes is protected against by the adaptive locking scheme described by Burgess and Skipitaris.[9]

INTERNAL FEEDBACK MECHANISMS

Many immune system functions occur as responses to detected threats. Cfengine classes which become defined manually, by partner systems, provide cooperative feedback to system administration, but classes must also be definable through an internal feedback mechanism. This ties in with the way cfengine compares the state of the system to a template or model and carries out repairs if necessary. When corrective actions are instigated, user-defined classes or flags may become switched on. This, in turn, switches on further checks or follow-up actions. For example, if

a file requires an update by copying, if diskspace exceeds specified limits, or if a file is edited, then user defined classes can be made to switch on, allowing subsequent modifiation to the program in order to follow up the change in the system. This can be used to good effect in garbage collection schemes by monitoring the free space on filesystems, and by switching on emergency tidying procedures when space falls below a certain threshold. This is just to cite just a single example. Another example would be the need to restart a daemon after its log files were rotated. Some daemons cannot write to their log files after a log rotation, since the file descriptor becomes stale.

Consider disk usage. The lower line of Figure 1 shows the filling of a student home directory disk, approximately 80 per cent full to begin with, plotted as a function of time. Without cfengine's regulatory actions, this disk would fill up more or less monotonically, since students never tidy their home directories willingly. The system would then be paralysed within a few days. Cfengine is programmed to clear core files, WWW cache files, compilation bi-products and backup files which have not been used for an appropriate period. Other files, such as zip files or mp3 sound files and even images which are not part of users' WWW pages, are deleted less frequently. This is local policy for student disks, not a fixed feature of cfengine. Cfengine tidies disks twice daily, once after lunch break and once during the night. In addition, tidying becomes active if the amount of free space on the disk volume falls under a specified limit. One can see from the graph how the disk filling factor falls abruptly as cfengine starts tidying. Abrupt rises in disk usage occur as users collect and unpack data from the internet. What this behaviour reflects is the need

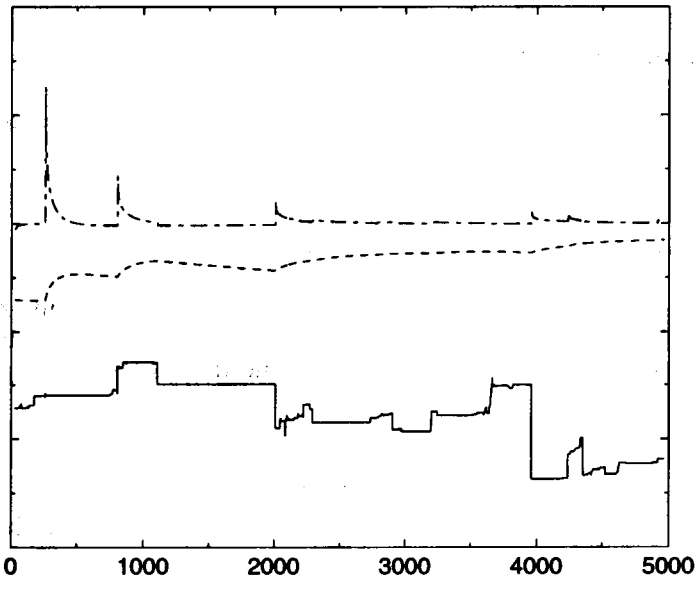

Figure 1. Disk usage as a function of time over the course of a week, beginning with Saturday. The lower solid line shows actual disk usage. The middle line shows the calculated entropy of the activity and the top line shows the entropy gradient. Since only relative magnitudes are of interest, the vertical scale has been suppressed. The relatively large spike at the start of the upper line is due mainly to initial transient effects. These even out as the number of measurements increases

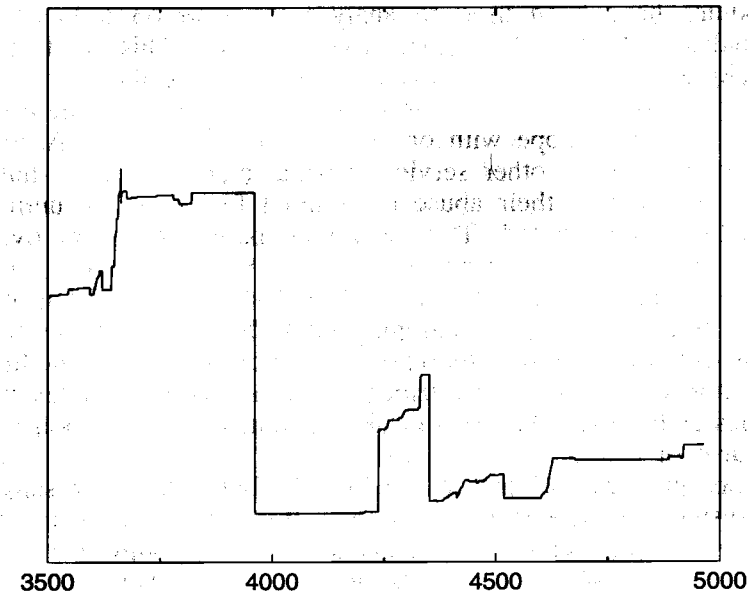

Figure 2. Disk usage as a function of time, magnification of the solid line of Figure 1. Sudden rises in student disk usage occur as users download or unpack data from the internet. Large rises typically occur around lunch breaks or after lectures when many users head for the terminal rooms

for large amounts of temporary filespace. By having cfengine clear temporary files regularly one avoids the need for hard disk quotas which could prevent a user from doing something as innocent as compiling a program, while at the same time keeping the level of disk usage under control for everyone.

EXTERNAL FEEDBACK MECHANISMS

Anticipating the move towards computer immunology, plug-in modules were added to the classing engine, so that custom analysis programs could be used to set abstract classes. Such modules are language independent and communicate by a simple textual protocol, not unlike CGI programs. Plug-in modules allow any user-supplied algorithm to build a model of the system's state, based on general criteria. Modules are added in a program's action sequence, i.e. that part of a cfengine program which indicates the order in which bulk operations are to be carried out, so that they can be called up at an appropriate moment, perhaps in response to a internal feedback. For security reasons modules must be placed in a specific directory and must follow a naming convention. Any classes which the module wants to define must be declared in this list.

```
actionsequence =
 (
 module: analyse-system.class1.class2
 files
 tidy
 )
```

Plug-in modules have been used to analyse the load on a WWW server and provide an analysis of most frequently accessed files. This module sets classes **http_low, http_medium** and **http_high** as activity rises above certain thresholds. These are used primarily as a warning at present, but could easily be used reorganize system resources to either cope with or resist the load in future. A similar module could be written to monitor other services so that cfengine could shut them down temporarily in the event of their abuse (spamming for instance), until a later time when they could be ressurected. This would be perceived as an overreaction by some, but if attacks occur at night when no-one is available to react, or if the server host is critical for other services, then this provides a very useful protection to the system and its disks. First aid is seldom perfect, but it saves lives and limbs. A milder reaction would be to renice the relevant processes; a more intelligent response would be to automatically add a malicious spam site to the mail exchanger's anti-spamming tables and notify a human for later examination. Local policy must dictate the correct course of action.

The above are just two off the cuff examples, but still rather standard in their approach to system management. A much more exciting idea is the long-term analysis of system trends. The plug-in module technology allows interprocess communication on a new level and opens cfengine to a world beyond the instant of its invocation. Previously missing from the classing mechanism was a coupling between program adaptability and long-term, statistical variations in the behaviour of the system. Some ailments are caused by a cumulative degradation of the system. Basic remedies are nonetheless be accessible to cfengine and can be activated by a battery of classes labelling appropriate countermeasures.

Statistical analysis is an important direction for the automated immunity of computer systems. Any complex system which develops over time can be analysed in terms of the average behaviour of its key variables. To some degree this is already reflected in programs which measure kernel performance (xload, perfmeter, etc.), but there is much more to be gained from time averages than plotting line graphs. The statistical interpretation of complex systems has been the realm of physics and mathematics for over a century. Some time analysis, or performance tuning, takes place in kernel, but this does not affect the system at the level of the system administrator.

There are many ways in which statistical information can be captured and used to extract and determine features of the system's behaviour. System statistics can be read both from the system logs and from kernel and filesystem audits. The only possible hindrance to the use of statistical analysis is the potential cost in terms of the resources required to compute the relevant metrics. These resources include CPU time and long-term disk storage of data.

QUANTITATIVE MEASURES FOR FEEDBACK

Entropy

As an example of what can be done with statistics, let us consider a simple cumulative quantifier of the system's history, namely its entropy or disorder. Entropy has certain qualitative, intuitive features which are easily understood. Disorder in a system measures the extent to which it is occupied by files and processes which

prevent useful work. If there is a high level of disorder, then – depending on the context – one might either feel satisfied that the system is being used to the full, or one might be worried that its capacity is nearing saturation. In the latter case, measures could be taken to even the load across the network.

To define a system entropy we need to look at the behaviour of the system over longer periods. There is no unique way of defining the entropy of the system, but in looking for an illustrative example we can steer clear of a number of definitions in which an administrative tool such as cfengine would not be interested. Disorder in kernel activity is not usually something which a configuration tool can do much to correct; these problems are best handled internally. We therefore restrict the realm of responsibility here to file and process usage.

Entropy is closely associated with the amount of granularity or roughness in our perception of information. Indeed, all statistical quantifiers are related to some procedure for coarse-graining information, or eliminating detail. In order to define an entropy one needs, essentially, to distinguish between signal and noise. This is done by blurring the criteria for the system to be in a certain state. As Shannon put it, we introduce redundancy into the states so that a range of input values (rather than a unique value) triggers a particular state. If we consider every single jitter of the system to be an important quantity, to be distingiushed by a separate state, then nothing is defined as noise and chaos must be embraced as the natural law. However, if one decides that certain changes in the system are too insignificant to distinguish between, such that they can be lumped together and categorized as a single state, then one immediately has a distinction between useful signal and error margins for useless noise. In physics. this distinction is thought of in terms of order and disorder.

Let us represent a single quantifier of system resources as a function of time $f(t)$. This function could be the amount of CPU usage, or the changing capacity of system disks for instance. We wish to analyse the behaviour of system resources by computing the amount of entropy in the signal $f(t)$. This can be done by coarse-graining the range of $f(t)$ into N predefined cells:

$$F^i_- < f(t) < F^i_+ \tag{1}$$

where $i = 1 \ldots N$,

$$F^i_+ = F^{i+1}_- \tag{2}$$

and the constants $F^i_\pm$ are the boundaries of the ranges (imagine drawing horizontal threshold lines from the tick marks of Figure 1, thus dividing up the graph into horizontal slices). The probability that the signal lies in the cell i, during the time interval from zero to T is the fraction of time the function spends in each cell i:

$$p_i(T) = \frac{1}{T}\int_0^T dt\,[\theta(f(t) - F^i_-) - \theta(f(t) - F^i_+)] \tag{3}$$

where $\theta(t)$ is the step function, defined by

$$\theta(t - t') = \begin{cases} 1 & t - t' > 0 \\ \frac{1}{2} & t = t' \\ 0 & t - t' < 0 \end{cases} \tag{4}$$

Now, let the statistical degradation of the system then be given by the Shannon entropy[12]

$$E(T) = -\sum_{i=1}^{N} p_i(T) \log p_i(T) \tag{5}$$

where p_i is the probability of seeing event i on average. i runs over an alphabet of all possible events from 1 to N, which is the number of independent cells in which we have chosen to coarse-grain the range of the function $f(t)$. The entropy, as defined, is always a positive quantity, since p_i is a number between 0 and 1.

Entropy is lowest if the signal spends most of its time in the same cell $F^i_{\pm}$. This means that the system is in a relatively quiescent state, and it is therefore easy to predict the probability that it will remain in that state, based on past behaviour. Other conclusions can be drawn from the entropy of a given quanitifier. For example, if the quantifier is disk usage, then a state of low entropy or stable disk usage implies little usage which in turn implies low power consumption. This might also be useful knowledge for a network; it is easy to forget that computer systems are reliant on physical constraints. If entropy is high it means that the system is being used very fully: files are appearing and disappearing rapidly; this makes it difficult to predict what will happen in the future and the high activity means that the system is consuming a lot of power. The entropy and entropy gradient of sample disk behaviour is plotted in Figure 1.

Another way of thinking about the entropy is that it measures the amount of noise or random activity on the system. If all possibilities occur equally on average, then the entropy is maximal, i.e. there is no pattern to the data. In that case all of the p_i are equal to $1/N$ and the maxmimum entropy is (log N). If every message is of the same type then the entropy is minimal. Then all the p_i are zero except for one, where $p_x = 1$. Then the entropy is zero. This tells us that, if $f(t)$ lies predominantly in one cell, then the entropy will lie in the lower end of the range $0 < E < \log N$. When the distribution of messages is random, it will be in the higher part of the range. This information can be used to reorganize system resources on a number of levels.

What can be seen from Figure 1 is that the rate of entropy change is greatest when the largest changes in disk usage take place. This usually signals periods of user behaviour when files are downloaded. Experience shows that these periods tend to occur in dense periods of activity, separated by quiet times. The load on the system NFS server becomes noticeable during these times and many trivial operations take a noticable length of time. At these times the system is vulnerable to saturation (a self-styled denial of service attack) and countermeasures are in order. The avenues of response are clearly site and policy dependent; examples include suspending ftp and www processes until disk activity falls below a safety threshold, as well as activating tidy operations to remove unnecessary files.

The definition of entropy given above leads to a cumulative result, as would any statistical survey of the system. This has several implications. The initial transient spike in Figure 1 becomes damped as time runs on, indicating an increasing tolerance of the metric to changes in behaviour. Computationally, also it would be impractical to retain a memory of past behaviour in great detail. The resources to store all of the data would be enormous. Some kind of compromise needs to be reached. A natural solution would be to reset counters each day or each week, since these are natural periods for usage. However, a complete reset would lose valuable information about previous patterns of behaviour. A project for future research would be to design a data collection system in which detailed data are kept only for a short time, but are regularly 'boiled down' to a summarial form which could be retained over very long periods, like a 'medical history' of the system.

The most efficient way of performing a statistical analysis is clearly to have a daemon collect data[13] about the system aynchronously. The data may then be stored in a convenient form. A cfengine plug-in module can quickly interrogate the accumulated data and modify the list of appropriate classes at cfengine's behest. Statistical averages distinguish themselves from quantities such as system calls in that they are real (floating point) numbers. The switching of classes based on statistical variables opens the way for cfengine to respond to fuzzy activation levels with perhaps several control thresholds. In most cases, the appropriate thresholds would have to be determined empirically, perhaps employing neural nets in the long-term. This is also an area for significant future research.

For the informational entropy to work usefully as a feedback mechanism, we need to be selective in the type of data which are collected. This is a key issue, since it is here that the coarse graining takes place. Statistics can be used in many ways; there are few absolutes so it is not the intention of this paper to suggest that profound answers are to be found from a simplified quantifier like the one described here. Looking for methods of analysis is a topic for a paper in its own right; suffice it to say that system analysis would be greatly facilitated by a standardized form of system reporting. Although unix systems have syslog, and a variety of 'stat' programs, these are not really adequate for the task, since there is no standard in the kinds of message which are produced. Syslog is more like a dumping ground for messages than it is an ordered registry. Also the question remains as to what else one might do to respond to a high level of entropy. There are many possibilities; the case considered here is only a simple example. One might, for instance, determine that peak entropy is caused by too many users and prevent further logins to the system by setting `/etc/nologin` or equivalent. Another possibility would be to simply add a login messages warning new users that the system is overloaded and would do well to go elsewhere. Yet another alternative would be to replace particularly demanding programs with links to a program which informs the user that the program is temporarily unavailable on the system concerned. Cfengine's adaptive locks use parameters which control their dead time and processes' 'time to live'. These parameters could also be altered by the feedback mechanisms to manipulate cfengine's ability to retain control of the system and protect itself from over-use (i.e. that cfengine itself becomes a problem rather than a help to the system). Clearly, there are many possibilities within the scope of cfengine's jurisdiction, but the dictionary of responses must be determined locally according to system policy.

What is important to realize is that system-load occupies a level of operation well

above low-level kernel activity. It is dependent on the behaviour of users, external boundary conditions, and on the physical limits of resources. An effective scheme which protects the system's ability to function adequately can only come about if one pays attention to this level of system operation. Traditionally, this has been improvised by system administrators. The aim here has been to automate it.

ALTERNATIVE METRICS

Entropy is not unique as a system metric. Another possibility would be the notion of long range order, or long-term stability of signal data. This is based on the idea of fluctuations in system metrics $v(t)$, represented by the correlation functions[14]

$$C_v(t,t') = \langle v(t)\, v(t')\rangle - \langle v(t)\rangle \langle v(t')\rangle \tag{6}$$

If the time series is correlated over long periods, it is essentially stable. Correlation functions have attractive mathematical features, but it could well be impractical to use them to follow the behaviour of the system in real time. Another approach would be to use neural networks and Fourier analysis to look for patterns in the time series. There might also be a future in attempting to model computer systems by dynamical mathematical systems. If suitable variables can be identified, one would expect a computer system to behave like a non-equilibrium field theory,[15,14] characterized by changing statistical potentials. A level of cross fertilization with statistical physics would be useful here. Threshold behaviour would be represented by different crossovers and phase transitions and the aim would be introduce a corrective force which preserves the system's linearity or stability to perturbations. Ironically, this is a return to analogue computing based on continuous potentials.

In any feedback system there is the possibility of instability: of either wild oscillation or exponential growth. Stability can only be achieved if the state of the system is checked often enough to adequately detect the resolution of the changes taking place. If the checking rate is too slow, or the response to a given problem is not strong enough to contain it, then control is lost. In biological systems such cases require medical assistance or they generally result in death. In computer systems, the result might be less disastrous but could lead to system paralysis and thrashing. These behaviours have been observed in the studies at Oslo, although not often.

REMARKS

Several variants of feedback regulation have been tested at Oslo in order to gauge the practical benefits of the foregoing ideas. Internal feedback based on disk usage is in permanent use since this can cure sudden cases of full disks, generally within under an hour even if no one is present and usually must faster. More complex metrics, such as entropy have been tested, with varying degress of success. Finding the right metrics for a system can be a difficult problem. There is probably no single metric which can summarize the behaviour of a system. For instance, databases can load disk activity heavily, but they might not lead to any significant variation in the amount of storage used, so the entropy example here would not capture that nuance of system behaviour. The entropy measurements described here have been

observed to capture certain social habits of users, however, such as when they download most of their files, or perform most of the web browsing, but even these conclusions are somewhat anecdotal and difficult to substantiate. One must be ready to accept that a given metric will depend on many interrelated factors. The example of a thermodynamical system which depends upon a number of metrics (temperature, pressure, volume, applied forces, etc.) is most likely the correct one. Thus trigger behaviour becomes not just a linear threshold, but a hypersurface threshold with complex regions.

The necessity of feedback regulation in system management is not in question. The mechanisms described in this paper can be used to automate system regulation to the extent whereby humans can be eliminated from the loop in a wide variety of situations. At Oslo, cfengine is responsible for nearly all system maintenance, short of installing and upgrading software. Simple feedback mechanisms such as the ones described here have proven highly successful in controlling the behaviour of the system; the question which remains is how one determines an appropriate policy of controls. The simplest extreme for preserving a system's well-being is to exclude all users. The opposite extreme of anarchy is equally absurd. Somewhere in between is an acceptable level of intervention, which must be tailored to the habits and needs of users on each network individually. In the final analysis, this must decided by a human.

It is encouraging that the basic concept of cfengine remains useful even when increased feedback is required. No doubt the details will be fine tuned further in the future and cfengine will cooperate with other automatic systems to provide a complete repertoire of immune reponses. What is important is the continual move towards not merely fault-tolerant, but fault-corrective systems.

REFERENCES

1. M. Burgess, 'Computer immunology', *Proc. 11th Systems Administration Conference (LISA).* (To appear 1998).
2. P. Anderson, 'Towards a high level machine configuration system', *Proc. 8th Systems Administration Conference (LISA)*, 1994.
3. M. Fisk, 'Automating the administration of heterogeneous LANS', *Proc. 10th Systems Administration Conference (LISA)*, 1996.
4. J. P. Rouillard and R. B. Martin, 'Config: a mechanism for installing and tracking system configurations', *Proc. 8th Systems Administration Conference (LISA)*, 1994.
5. J. Finke, 'Automation of site configuration management', *Proc. 11th Systems Administration Conference (LISA)*, 1997, p. 155.
6. M. Burgess, 'A site configuration engine', *Computing Systems*, **8**, 309 (1995).
7. M. Burgess and R. Ralston, 'Distributed resource administration using cfengine', *Software—Practice and Experience*, **27**, 1083 (1997).
8. R. Evard, 'An analysis of unix system configuration', *Proc. 11th Systems Administration Conference (LISA)*, 1997, p. 179.
9. M. Burgess and D. Skipitaris, 'Adaptive locks for frequently scheduled tasks with unpredictable runtimes', *Proc. 11th Systems Administration Conference (LISA)*, 1997, p. 113.
10. V. Paxson, 'Bro: A system for detecting network intruders in real time', *Proc. 7th USENIX Security Symposium*, 1998.
11. S. E. Hansen and E. T. Atkins, 'Automated system monitoring and notification with swatch', *Proc. 7th Systems Administration Conference (LISA)*, 1993.
12. C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication*, University of Illinois Press, 1949.
13. R. Emmaus, T. V. Erlandsen and G. J. Kristiansen, 'Network log analysis', *Oslo College dissertation*, Oslo, 1998.