Computer Immunology

Mark Burgess, Oslo College

Computer Immunology

Mark Burgess – Oslo College

ABSTRACT

Present day computer systems are fragile and unreliable. Human beings are involved in the care and repair of computer systems at every stage in their operation. This level of human involvement will be impossible to maintain in future. Biological and social systems of comparable and greater complexity have self-healing processes which are crucial to their survival. It will be necessary to mimic such systems if our future computer systems are to prosper in a complex and hostile environment. This paper describes strategies for future research and summarizes concrete measures for the present, building upon existing software systems.

Autonomous systems

We dance for our computers. Every error, every problem that has to be diagnosed schedules us to do work on the system's behalf. Whether the root cause of the errors is faulty programming or simply a lack of foresight, human intervention is required in computing systems with a regularity which borders on the embarrassing. Operating system design is about the sharing of resources amongst a set of tasks; additional tasks need to be devoted to protecting and maintaining a computer with an *immune system* so that human intervention can be minimized.

Imagine what the world would be like if humans were as helpless as computer systems. Doctors would be paged every time a person felt unwell or had to do something as basic as purge their waste 'files.' They would then have to summon the person concerned in order to perform the necessary dialysis procedures and push pills into their mouths manually. Fortunately most humans have self-correcting systems which work both proactively and retroactively to prevent such a situation from arising. Not so computers: it is as though all of our machines are permanently in hospital.

This paper is about the need for a new paradigm leading to the construction of a bona fide computer immune system. With an immune system, a computer could detect problem conditions and mobilize resources to deal with them automatically, letting the machine do the work. Although the phrase 'immune system' would make many people think immediately of computer viruses, there is much more to the business of keeping systems healthy than simply protecting them from attack by hostile programs. If one thinks of biological systems or other self-sufficient systems, such as cities and communities, some of the most critical subsystems are involved in cleaning up waste products, repairing damage and security through checking and redundancy. It would be unthinkable to do without them.

Surprisingly most system administration models which are developed and sold today are entirely based either on the idea of interaction between administrator and either user or machine; or on the cloning of existing systems. We see user graphical user interfaces of increasing complexity, allowing us to see the state of disarray with ever greater ulcer-provoking clarity, but seldom do we find any noteworthy degree of autonomy. In other words administrators are being placed more and more in the role of janitors or doctors with pagers. We are giving humans more work, not less.

The aim here is to promote serious discussion and research activity in the area of autonomic system maintenance. System administration overlaps with so many other areas of computing that it is generally forgotten as a side issue by the academic community. I would like to argue that it is one of the most pressing issues that we face. Dealing with the complexity of the network is the main challenge of the next century. Every multiprocess computer system is already a micro-cosmic virtual network. Computer resources have perhaps been too precious to make defensive or preventative systems feasible before now (and we have been distracted by other more glamorous issues), but the time is right to build not merely fault tolerant systems, but self-maintaining, fault-corrective systems. In the sections which follow I would like to explore this idea and discuss how one might efficiently build such systems.

Historical

The idea of self maintaining computer systems is not new but, as with many modern technologies (telecommunications, robotics), it originates in science fiction rather than science fact. There are dozens of examples of autonomous systems in speculative writings. The artificial intelligence community has been developing analogous systems using techniques developed over the past thirty years; some of these have even been used to create diagnostic systems for human beings, but not computers.

In 1974, science fiction writer John Brunner wrote *Shockwave Rider* [1], building on Alvin Toffler's *Future shock*. In his world of fax machines, laser printers, laptops and mobile phones, where governments argue about the public freedom to encrypt data, we find computer worms which propagate across the equivalent of the Internet performing vital (and non-vital) services quite autonomously. In this world, most computing transactions occur by creating worms, or intelligent agents which work in the background on behalf of users. Such is the extent of these worms that operating systems are necessarily programmed to give them a low priority to avoid being swamped (spammed). This is something which we experience today. His solution is correct but too simplistic for a real world system. A full immune system would need to be less passive. It was, incidentally, only a few years later in 1988 that the first Internet worm (propagating infectious agent) was thrust to the forefront of our attention [2]. Even earlier examples of autonomous systems include Robbie the robot in Forbidden Planet and HAL in the film 2001: A space odyssey. HAL was a self diagnosing, but not self-repairing system, but he was also guilty of mobilizing human power and even sending them on wild goose chases, to fix a problem which could almost certainly have been dealt with automatically!

There are several valuable insights to be made by comparing computing systems to biological and social systems. Biological and social systems have solved most of the problems of self-sufficiency with ingenious efficiency. Science fiction writers too have expended many pages exploring the consequences of speculative ideas. It is not merely for whimsical amusement that such comparisons are valuable. All ideas should be considered carefully, particularly when they are based on millions of years of evolution or a hundred years of reflection.

Mechanical robots manage removable storage media even today. Robots which repair computer hardware are experimented with in England, but software robots – artificial agents which perform manual labour at the system level – are almost non-existent. One exception is cfengine [3, 4, 5, 6], a software robot which can sense aspects of the state of the system and alter its program accordingly. Cfengine can perform rudimentary maintenance on files and processes, but it is at the lower threshold of intelligence on the evolutionary scale. A system like cfengine will be the hands or manipulators of our future systems, but more complex recognition systems are needed to select the best course of action. Cfengine is not so much as robot as it is a claw.

One of science-fiction's common scenarios is that machines will run amok and turn against humankind. In a sense, bug ridden software does just this today, and system crackers write programs which corrupt the behavior of the system so as to attack the user. Isaac Asimov's answer to this problem, developed in detail in the 1940's, was to endow automatons with a set of rules which curbed their behavior and prevented them from harming humans. In a sense this was a theoretical immune system.

1. *A robot may not injure a human being, or through inaction allow a human being to come to harm.*
2. *A robot must obey the orders given to it by human beings, except where such orders would conflict with the First Law.*
3. *A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.*

These rules are more than nostalgia; there is a serious side to building the analogue of Asimov's laws of robotics [7] into operating systems. If one replaces the word robot with system and human with user, it seems less fanciful. The practical difficulty is to translate whimsical words into concrete detectable states. This starts to sound like artificial intelligence, but a less intensive solution might also be possible. In fact, such basic rules already work as a loose umbrella for the way in which systems work, but that looseness can be tightened up and made into a formal protocol. As Asimov discussed in the forties, the potential for human abuse of systems which are required to follow rigid programs is great. The system vandals of the future will have new rule systems to exploit in their pursuit of mischief. Our task is to make the rules and protocols of the future as immune as possible to corruption. This is only possible if those rules present a moving target, i.e., we aim for adaptive systems.

As a chemist, Asimov based his robots on analogue computing technology with varying potentials, not unlike the behavior of the body. In modern jargon they were based on fuzzy logic. Digital systems abandoned analogue computing long ago, but there is still a statistical truth in such analogue notions. Continuous variables may yet replace the digital logic of our canonical programming paradigms in a wide range of applications, not as analogue electrical potentials but as statistical or thermodynamical average potentials. Quantities analogous to physical variables temperature and entropy can be defined on the basis of the average behavior of computer systems. Such variables act as book keeping parameters and could be used to simplify and make running sense of system logs, for example. In his Foundation books, a statistical theory of society called *psycho-history* was proposed. The reality of this may be observed in the present day statistical mechanics of complex physical and biological systems (including immunology) as well as in weather forecasting. The statistical analysis of complex systems in the natural world is a science which is presently being constructed from origins in statistical physics. It will most likely converge with the work in pattern recognition and neural computing. Computer immunology needs to be there alongside its biological counterpart.

A lot of research work has been devoted to the development of mechanical robots, in the areas of pattern recognition and expert systems, but at the bottom of all of these lies a computer system which makes humans subservient to its failures.

Present Day Solutions

Present day computer systems are not designed with any sophisticated notions of immunity in mind, but most of them are flexible enough to admit the integration of new systems. How far could we go in constructing an immune system today, even as an afterthought? Many proponents of automation have built systems which solve specific problems. Can these systems be combined into a useful cooperative? The LISA conferences have reported many ideas for automating system administration [8, 9, 10, 11]. Most of these have been ways of generating shell or perl scripts. Some provide ways of cloning machines by distributing files and binaries from a central repository.

Cfengine on the other hand is a tool, written by the author, which differs from previous systems in a number of ways. Firstly it does not use linear, procedural programming such as shell or perl, it is a much higher level descriptive language. The second difference is that it has converging semantics, i.e., one describes what a system should look like, and when the system has been brought to that state, cfengine becomes inert. A third point about cfengine is that its decision making process is based on abstract classes which allows for more powerful administration models than we have traditionally been used to. Finally it offers protection against unfortunate repetition of tasks and hanging processes in situations where several administrators are working independently with little opportunity to communicate [12]. Cfengine was designed with computer immunity in mind.

In spite of the enormous creative effort spent developing the above systems, few if any of them will survive in their present form in the future. As indicated by Evard in a presentation at LISA 1997 [13] analyzing many case studies, what is need now is a greater level of abstraction. Although its details are not yet optimal, the idea behind cfengine is basically sound and meets most of Evard's requirements, but even this will not survive in present form. It is built as a patch for our present operating systems. Ideally such a system would be built into the core of a modern operating system. The present Unix model is in need of an overhaul: even a small one would help significantly.

Corrective systems are not the only way in which one can improve present day computers. Network services are a mixture of uncoordinated mechanisms, using `inetd` or `listen` to start heavyweight processes, or based on permanently listening daemons. An interesting model which could replace these tools is the ACE system [14]. ACE (the Adaptive Communication Environment) is an extensive base of C++ classes which provide the necessary paradigms for network services in neatly packaged objects. ACE can use lightweight processes (threads) or heavyweight processes, and can load classes on the fly in order to optimize the servicing of network protocols. ACE is well structured and carefully crafted, even though it attempts to straddle and conceal the differences between diverse Unix systems and NT. This kind of modular approach could be used to strengthen network reliability and security.

Many projects now under development, could help to improve the state of off-the-shelf operating systems. It will be up to system designers to adopt these as standards. The challenge is to compress a protective scheme into low overhead threads which will not noticeably affect system performance during peak usage. The intervention of a human should be as far as possible avoided.

Management Tools

The main focus in system administration today is in the development of man-machine interfaces for system management.

Tivoli [15] is a Local Area Network (LAN) management tool based on CORBA and X/Open standards; it is a commercial product, advertised as a complete management system to aid in both the logistics of network management and an array of configuration issues. As with most commercial system administration tools, it addressed the problems of system administration from the viewpoint of the business community, rather than the engineering or scientific community. It encourages the use of IBM's range of products and systems, and addresses other widely used systems through its use of open standards. Tivoli's most important feature in the present perspective is that it admits bidirectional communication between the various elements of a management system. In other words, feedback methods could be developed using this system. The apparent drawback of the system is its focus on application level software rather than core system integrity. Also it lacks abstraction methods for coping with with real world variation in system setup.

HP OpenView [16] is a commercial product based on SNMP network control protocols. OpenView aims to provide a common configuration management system for printers, network devices, Windows and HPUX systems. From a central location, configuration data may be sent over the local area network using the SNMP protocol The advantage of OpenView is a consistent approach to the management of network services; its principal disadvantage, in the opinion of the author, is that the use of network communication opens the system to possible attack from hacker activity. Moreover, the communication is only used to alert a central administrator about perceived problems. No automatic repair is performed and thus the human administrator is simply overworked by the system.

Sun's Solstice [17] system is a series of shell scripts with a graphical user interface which assists the administrator of a centralized LAN, consisting of Solaris machines, to initially configure the sharing of printers, disks and other network resources. The system is basically old in concept, but it is moving towards the ideas in HP OpenView.

Host Factory [18] is a third party software system, using a database combined with a revision control system [19] which keeps master versions of files for the purpose of distribution across a LAN. Host Factory attempts to keep track of changes in individual systems using a method of revision control. A typical Unix system might consist of thousands of files comprising software and data. All of the files (except for user data) are registered in a database and given a version number. If a host deviates from its registered version, then replacement files can be copied from the database. This behavior hints at the idea of an immune system, but the heavy handed replacement of files with preconditioned images lacks the subtlety required to be flexible and effective in real networks. The blanket copying of files from a master source can often be a dangerous procedure. Host Factory could conceivably be combined with Cfengine in order to simplify a number of the practical tasks associated with system configuration and introduce more subtlety into the way changes are made (it is not always necessary to replace an arm in order to remove a wart). Currently Host Factory uses shell and Perl scripts to customize master files where they cannot be used as direct images. Although this limited amount of customization is possible, Host Factory remains essentially an elaborate cloning system.

In recent years, the GNU/Linux community has been engaged in an effort to make Linux (indeed Unix) more user-friendly by developing any number of graphical user interfaces for the system administrator and user alike. These tools offer no particular innovation other than the novelty of a more attractive work environment. Most of the tools are aimed at configuring a single stand-alone host, perhaps attached to a network. Recently, two projects have been initiated to tackle clusters of Linux workstations [20, 21].

While all of the above tools fulfill a particular niche in the system administration market, they are basically primitive one-off configuration tools, which lack continuous monitoring of the configuration. It would be interesting to see how each of these systems handled the intervention of an inexperienced system administrator who, in ignorance of the costly software license, meddled with the system configuration by hand. Would the sudden deviation from the system model lead to incorrect assumptions on the part of the management systems? Would the intervention destroy the ability of the systems to repair the condition, or would they simply fail to notice the error? In most cases, it is likely that all three would be the result. The lack of continuous assessment is a significant weakness.

Monitoring Tools

Monitoring tools have been in proliferation for a number of years [22, 23]. They usually work by having a daemon collect some basic auditing information, setting a limit on a given parameter and raising an alarm if the value exceeds acceptable parameters. Alarms might be sent by mail, they might be routed to a GUI display or they may even be routed to a system admin's pager [23].

The network monitoring school has done a substantial amount of work in perfecting techniques for the capture and decoding of network protocols. Programs such as `etherfind`, `snoop`, `tcpdump` and Bro [24] as well as commercial solutions such as Network Flight Recorder [25] place computers in 'promiscuous mode' allowing them to follow the passing data-stream closely. The thrust of the effort here has been in collecting data, rather than analyzing them in any depth. The monitoring school advocates storing the huge amounts of data on removable media such as CD to be examined by humans at a later date if attacks should be uncovered. The analysis of data is not a task for humans however. The level of detail is more than any human can digest and the rate of its production and the attention span and continuity required are inhuman. Rather we should be looking at ways in which machine analysis and pattern detection could be employed to perform this analysis – and not merely after the fact. In the future adaptive neural nets and semantic detection will be used to analyze these logs in real time, avoiding the need to even store the data.

An immune system needs to be cognizant of its local host's current situation and of its recent history; it must be an expert in intrusion detection. Unfortunately there is currently no way of capturing the details of every action performed by the local host, in a manner analogous to promiscuous network monitoring. The best one can do currently is to watch system logs for conspicuous error messages. Programs like SWATCH [23] perform this task. Another approach which we have been experimenting with at Oslo college is the analysis of system logs at a statistical level. Rather than looking for individual occurrences of log message, one looks for patterns of logging behavior. The idea is that, logging behavior reflects (albeit imperfectly) the state of the host [26].

Fault Tolerance and Redundancy

Fault tolerance, or the ability of systems to cope with and recover from errors automatically, plays a special role in mission critical systems and large installations, but it is not a common feature of desktop machines. Unix is not intrinsically tolerant, nor is NT, though tools like cfengine go some way to making them so. In order to be fault tolerant a system must catch exceptions or perform preliminary work to avoid fault occurrence completely. Ultimately real fault tolerance must be orchestrated as a design feature: no operation must be so dependent on a particular event that the system will fail if it does not occur as expected.

One of the reasons why large social and biological systems are immune to failure is that they possess an inbuilt parallelism or redundancy. If we scrape away a few skin cells, there are more to back up the missing cells. If we lose a kidney, there is always another one. If a bus breaks down in a city, another will come to take its place: the flow of public transport continues. The crucial cells in our bodies die at a frightening rate, but we continue to live and function as others take over. component is very important.

Fault tolerance can be found in a few distributed system components [27]: in file-systems like AFS and DFS [28]. Disk replication and caching assures that a backup will always be available. RAID strategies also provide valuable protection for secondary storage [29]. At the process level one has concepts such as multi-threading and load balancing. Experimental operating systems such as Plan 9 [30] and Amoeba [31, 32] are designed to be resistant to the performance of a single host by distributing processes transparently between many cooperating hosts in a seamless fashion. Fault tolerance in Arjuna [33] and Corba [34] is secured in a similar way.

Ideally however we do not want fault tolerant systems but systems which can correct faults once they have occurred. Faults are inevitable: they are something to be embraced, not swept under the rug. Some work has been done in this area in order to develop software reliability checks [35], but the reliability of an entire operating system relies not only on individual software quality but also on the evolution and the present condition of the system in its entirety. It is impossible to deal with every problem in advance. Presently computer systems are designed and built in captivity and then thrown, ill-prepared, into the wild.

Feedback Mechanisms: cfengine

Cfengine [3, 4, 36] fulfills two roles in the scheme of automation. On the one hand it is an immediate tool for building expert systems to deal with large scale configuration, steered and controlled by humans. It simplifies a very immediate problem, namely how to fix the configuration of large numbers of systems on a heterogeneous network with an arbitrary amount of variety in the configuration. On the other hand, cfengine is also a significant component in the proposed immunity scheme. It is a phagocyte which can perform garbage collection; it is a drone which can repair damage and build systematic structures.

A reactor, or event loop, is a system which detects a certain condition or signal and activates a response. Reactor technology has penetrated nearly all of the major systems on which our networks are based. It is at the center of the client-server model, and windowing technology. It is a method of making decisions in a dynamical and structured fashion. Reactors must play a central role in computer immunity.

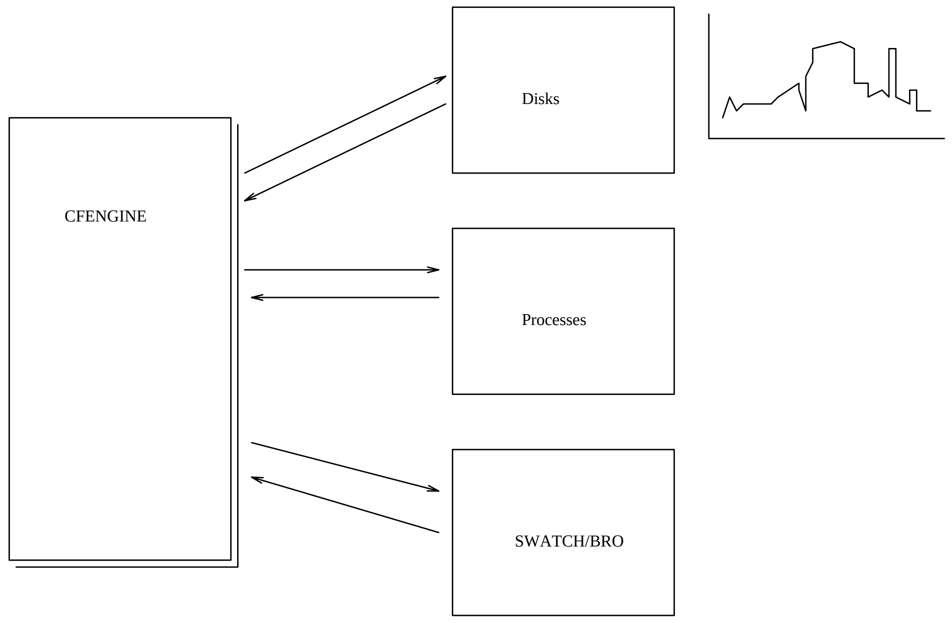

Figure 1: Cfengine communicates with its environment in order to stabilize the system. This communication is essential to cfengine's philosophy of converging behavior. Once the system is in the desired state, cfengine becomes quiescent.

Several systems are already based on this idea. Cfengine is a reactor which works by examining the state of a distributed computer network and switching on predefined responses, designed to correct specific problems. SWATCH [23] is a reactor which looks for certain messages in system log files. On finding particular messages it will notify a human much more visibly and directly than the original log message. In this respect SWATCH is a filter/amplifier or signal cleaning tool. See Figure 1.

Reactors lead naturally to another idea: that of back-reaction, or feedback [37]. If one system can respond to a change in another, then the first system should be able to re-adapt to the changes brought about by the second system, forming a loop. For instance, cfengine can examine the state of a Unix system and run corrective algorithms. Now suppose that cfengine logs the changes it makes to the system so that the final state is known. These changes could then be re-analyzed in order to alter cfengine's program the next time around. In fact, anticipating the need for cooperative behavior, cfengine already has the necessary mechanisms to respond to analyses of the system: if its internals are insufficient, plug-in modules can be used to extend its capabilities. This interaction with modules allows cfengine to communicate with third party systems and act back on itself, adapting its program dynamically in response to changes in its perceived classification of the environment. This is essential to cfengine's converging semantics.

At the network level, the same idea could be applied to dynamical packet filtering or rejection. Network analyses based on protocols can be used to detect problem conditions and respond by changing access control lists or spam filters accordingly. The mechanisms for this are not so easily implemented today since much filtering takes place in routers which have no significant operating system, but adaptive firewalls are certainly a possibility.

Security

Security is a thorny issue. Security is about perceived threats: it is subjective and needs to be related to a security policy. This sets it on a pedestal in a general discussion on system health, so I do not want to discuss it here. Nevertheless, a healthy system is inherently more secure by any definition and security can, in principle, be dealt with in a similar manner to that discussed here [38, 39]. Since network security is very much discussed at present [40, 39, 41, 42, 43, 44], I focus only on the equally important but much more neglected issue of stability.

Biological and Social Immunity

The body's immune system deals with threats to the operation of the body using a number of pro-active and reactive systems. We can draw important lessons and inspiration from the annals of evolution. There are three distinct processes in the body: those which fight infection, those which purge waste products and those which repair damaged tissues.

Prevention

The first line of defense against infectious disease in the body is the skin. The skin is a protective fatty layer in which most bacteria and viruses cannot survive. The skin is the body's firewall or viral gatekeeper, and with that we need not say more. The stomach and gut are also well protected by acid. Only one in ten million infectious proteins entering the body orally actually penetrate into the interior, most are blocked or broken down in the gut.

Another mechanism which prevents us from poisoning ourselves is the cleansing of waste products and unwanted substances from the blood. Natural killer cells, phagocytes and vital organs cooperate to do this job. If the blood were not cleansed regularly of unwanted garbage, it would soon be so full of cells that we would suffocate and our veins and arteries would clog up. In a similar way we can compare the performance of a system with and without a tool, like cfengine, to carry out essential garbage collection. In social systems, buildings or walls keep incompatible players apart. In computing systems one has object classes and segmentation to perform the same function. *Ultimately computer systems need to learn to distinguish illness from health, or good from bad. If such criteria can be defined in terms of computable states and policies, then illness prevention can be automated and an immune system can be built.*

Infection or Attack

When the body is infected or threatened, it mobilizes cells called lymphocytes (or B and T cells) to deal with the threat. 'Antigens', (antibody-generating threats) are often thought of as entities which are foreign to the body, but this is not necessarily the case. Complex systems are quite capable of poisoning themselves.

Normally cells in the body die by mechanisms which fall under a category known as *programmed cell death* (apoptosis). In this case, the cells remain intact but eventually cease to function and shrivel up (analogous to death signalled by the child-done signal SIGCHLD). Cells attacked by an infection die explosively, releasing their contents (analogous to a segmentation fault signal SIGSEGV), including proteins into the ambient environment. This is called *necrosis*. In one compelling theory of immunology, this unnatural death releases proteins into the environment which signal a crisis and activate an immune response. This provides us with an obvious analogy to work with.

The immune system comprises a battery of cells in almost every bodily tissue which have evolved to respond to violent cell death, both fighting the agents of their destruction and cleaning up the casualties of war: B-cells, T-cells, macrophages and dendritic cells to name but a few. Antigens are cut up and presented to T cells. This activates the T cells, priming them to attack any antigens which they bump into. B-cells secrete antibody molecules in a soluble form. Antibodies are one of the major protective classes of molecules in our bodies. Somehow the immune system must be able to identify cells and molecules which threaten the system and distinguish them from those which *are* the system. An important feature of biological lymphocytes is the existence of receptor molecules on their surfaces. This allows them to recognize cellular objects. Recognition of antigen is based on the complementarity of molecular shapes, like a lock and key.

The canonical theory of the immune system is that lymphocytes discriminate between self and non-self [45, 46, 47, 48, [49] (part of the system or not part of the system). This theory suffers from a number of problems to do with how such a distinction can be made. Foreign elements enter our bodies all the time without provoking immune responses, for instance during eating and sex. The body has its own antigens to which the immune system does not respond. This leads one to believe that self/non-self discrimination as a human concept can only be a descriptive approximation at best; from a computer viewpoint it would certainly be a difficult criterion to program algorithmically. Recent work on the so-called *danger model* [50] proposes that detector cells notice the shrapnel of non-programmed cell death and set countermeasures in motion. A dendritic cell attached to a body cell might become activated if the cell to which it is attached dies; the nature of the signalling is not fully understood. Although controversial, this theory makes considerable sense algorithmically and suggests a useful model for computer immunity: signals are something we know how to do.

The immune system recognizes something on the order of 10^7 different types of infectious protein at any given time, although T-cells have the propensity to detect a repertoire of 10^{16} and B-cells 10^{11} [51]. Apart from being a remarkable number to contemplate, the way nature accomplishes this provides some ingenious clues as to how an artificial immune system might work. The immune response is not 100 percent efficient: it does not recognize every antigen with complete certainty. In fact it is only something on the order of 10^{-5} or 0.001 percent efficient. What makes it work so effectively, in the face of this inefficiency, is the large number of cells in circulation (on the order of 10^{12} lymphocytes). There is redundancy or parallelism in the detection mechanism. Since the cells patrolling the body for invaders rely on spot checks, it is necessary to compensate for the contingency of failure by making more checks. In other words, the body does not set up roadblocks which check every cell's credentials: it relies on frequent random checks to detect threats. Indeed, there would not be room in the body for a fighting force of cells to match every contingency so new armies must be cloned once an infection has been recognized. The dead or marked cells are consumed by the body's garbage collection mechanism: macrophages 'eat' any object marked with an antibody. Phagocytes are the cells which engulf dead cells and remove them from the system.

Originally it was believed [52] the body was able to manufacture antibody only after having seen invading antigens in the body. However later it was shown [53, 54] that the body can make antibody for an antigen which has never existed in the history of the world. Having a repertoire with predetermined (random) shapes, the body uses a method of Darwinian (clonal) selection. Cells which are recognized proliferate at the expense of the rest of the population. The computer analogy would be to create a list of all possible checks and to change the priority of the checks in response to registered attacks. Seldom used attacks migrate down the list as others rise to the forefront of attention. This is also closely related to neural behavior and suggests that neural computing methods would be well suited to the task. Learning in a neural net is accomplished by random selection provided there is a criterion of value which selects one neural pathway over another when the correct random pathway is selected. In the case of a learning baby making random movements to grasp objects, the (presumably genetically inherited) criterion is the 'pleasure of success' in targeting the objects. In building a system of automatic immunity based on cheaply computed principles, it the basic criteria for good and evil, or healthy and sick which must be determined first.

The message is this: autonomous systems do not have to be expensive provided the system holds down the number of challenges it has to meet to an acceptable level. In the body, the immune system does not maintain a huge military presence in the body at all times. Rather it has a few spies which are present to make spot checks for infection. The body clones armies as and when it needs them. Inflammation of damaged areas signals increased blood flow and activity and ensures a rapid transport of cloned killer cells to the affected area as well as a removal of waste products. In a similar way, computers could alter their level of immune activity if the system appeared abnormal. Balance through feedback is important though: cancer is one step away from cloning.

Biological protocols

Protocols, or standards of behavior, are the basic mechanism by which orderliness and communication are maintained in complex systems. In the body a variety of protocols drive the immune system. The immune system encounters intruders via a battery of elements: antibody markers, T-cell presenting cells, the Peyers patches in the gut and so on [55]. In social systems one has rules of behavior, such as: put out the garbage on Tuesdays and Fridays; put money in the parking meter to avoid having your car swallowed by some uniformed phagocyte and so on.

Presently the main protocol for dealing with failure in the computing world is the 2001 syndrome [56]: wait for the system to collapse and then fix it. Complete collapse followed by reboot. No other organization or system in the world functions with such a singular disregard for its own welfare and the welfare of its dependents (the users). If a light bulb burns out and we replace it, there is no significant loss to its dependents. If a computer crashes users can lose valuable data, not merely time. Protocol solutions need to be built into the fabric of operating systems.

Computer Immunity and Repair

Computer Lymphocytes

What can we adapt from biological systems in order to build not merely fault tolerant systems, but fault correcting systems? Are the mechanisms of natural selection and defensive counter attack useful in computer systems?

The main difference between a computer system and the body is that the numbers are so much smaller in a computer system that the discrete nature of the system is important. Pattern recognition is a useful concept, but how should it be applied? The recognition of patterns in program code could be applied to individual binaries and might be used to detect potentially harmful operations, such as programs which try to execute "rm -rf *" or which attempt to conceal themselves using standard tricks. In order to select programs-to-allow and programs-to-reject one must search for code strings which can lead to dangerous behavior.

Self/non-self is not a very useful paradigm in computing and some immunologists believe that it is also an erroneous concept in biology. It is clearly irrelevant where a program originates; indeed we are actively interested in obtaining software from around the world. Such transplants or implants are the substance of the Internet. Rather, we could use a *danger model* [50] to try to detect programs which exhibit dangerous behavior as they run. The danger model in biology purports that the immune system responds to chemical signals which are leaked into the environment through the destruction of attacked cells. In other words, it is things which cause damage which activate an immune response. Here we shall define a danger model to be one in which an immune response is based on the general detection of dangerous conditions in the system. An immune system lies dormant until a problem is detected and wakes up in response to some signal of damage. This is the opposite of the way a firewall works, or preventative philosophies such as the security model the Java virtual machine [57]. In an immune system one already admits the defeat of prevention.

Today, the necessary danger signals might be found in the logs of programs running in user mode, or from the kernel exec itself. Ideally programs would not just log alerts to syslog, they would be able to activate a response agent (a lymphocyte) to fight the infection, i.e., the logging mechanism would be a reactor like inetd or listen, not just a dumb receptor [23].

A more efficient danger model for the future could be constructed by introducing a new standardized signalling mechanism. If each system process had a common standard of signalling its perceived state (in addition to, and different from, the existing signals.h) this could be used to calculate a vector describing the collective state of the system. This could, in turn, be used to create advanced feedback systems, discussed in the fourth section. To diagnose the correct immune response, programs need to be able to signal their perceived state to the outside world. Normally this is only done in the event of some catastrophe or on completion, but computer programs are proportionally more valuable to a computer than cells are to the body and we are interested in the effect each program has on the totality of the system. A program is often in the best position to know what and when something is going wrong. Outside observers can only guess. In some ways this is the function of system logs today, but the information is not in a useful form because it is completely non-standard and cannot be acted upon by the kernel or an immune system. To provide the simplest picturesque example of this kind of signalling, consider the characterization of running processes by the basic 'emotional' states or the system weather:

- Happy/Sunny (Plenty of resources, medium activity)
- Sad/Cloudy (Low on resources)
- Surprised/Unsettled (System is not in the state we expect, attack in progress, danger?)
- Angry/Stormy (System is responding to an attack)

Using such insider information, an immune response could be switched on to counter system stress. In order to be effective in practice, such states need to be related to a specific resource, for example: disk requirements, CPU requirements, the number of requests waiting in event queues etc. This would allow the system to modify its resource allocation policies, or initiate countermeasures, in order to prevent dangerous situations from developing. It is tempting to think of processes which could quickly pin-point the source of their troubles and obtain a response from the immune system, but that is a difficult problem and it might prove too computationally expensive in practice. Since the system kernel is responsible for resource allocation, such a scheme would benefit from a deep level of kernel cooperation. A graded signal system would be a good measure of system state, but it needs to be tied to resource usage in a specific way. See also reference [29].

Assuming that such a signalling model were implemented, how would counter-measures be initiated? In the body there are specific immune responses and non-specific immune responses. If we think in terms of what an existing cfengine based immune system could do to counter stressed systems there are two strategies: we could blindly start cfengine with its entire repertoire of tests and medicines to see if thrashing in the dark helps, or we could try to detect and activate only particular classes within a generic cfengine program to provide a specific response. These are also essentially the choices offered by biology.

The detection of dangerous programs by the effect they have on system resources is a 'danger model.' A self-non-self model based purely on recognition requires the identification and verification of program entities. This would be computationally inefficient. New in-coming programs would have to be analyzed with detection algorithms. Once verified a program could be marked with an encryption key signature for its authenticity to prevent the immune system from repeating its lengthy analysis. Or conversely, dangerous programs could be labelled with 'antibody' to prevent them from being used. Cfengine recognizes this kind of philosophy with its 'disable' strategy of rendering programs non-executable, but it requires them to be named in advance.

What is impressive about the biological immune system is that it recognizes antigens which the body has never even seen before. It does not have to know about a threat in order to manufacture antibody to counter it. Recognition works by jigsaw pattern-identification of cell surface molecules out of a generic library of possibilities. A similar mechanism in a computer would have to recognize the 'shapes' of unhealthy code or behavior [58, 59]. If we think of each situation as begin designated by strings of bytes, then it might be necessary to identify patterns over many hundreds of bytes in order to achieve identify a threat. A scaled approach is more useful. Code can be analyzed on the small scale of a few bytes in order to find sequences of machine instructions (analogous to dangerous DNA) which are recognizable programming blunders or methods of attack. One could also analyze on the larger scale of linker connectivity or procedural entities in order to find out the topology of a program.

To see why a single scale of patterns is not practical we can gauge an order of magnitude estimate as follows [51]. Suppose the sum of all dangerous patterns of code is S bytes and that all the patterns have the same average size. Next suppose that a single defensive spot-check has the ability to recognize a subset of the patterns in some fuzzy region ΔS, i.e., a given agent recognizes more than one pattern, but some more strongly than others and each with a certain probability. Assume the agents are made to recognize random shapes (epitopes) that are dangerous, then a large number of such recognition agents will completely cover the possible patterns. The worst case is that in which the patterns are randomly occurring (a Poisson distribution). This is the case in biology since molecular complexes cannot process complex algorithms, they can only identify affinities. With this scenario, a single receptor or identifier would have a probability of $\frac{\Delta S}{S}$ of making an identification, and there would be a probability $1 - \frac{\Delta S}{S}$ of not making an identification, so that a dangerous item could slip through the defenses. If we have a large number n of such pattern-detectors then the probability that we fail to make an identification can be simply written,

$$P_n = (1 - \frac{\Delta S}{S})^n \approx e^{-n\frac{\Delta S}{S}}.$$

Suppose we would like 50% of threats to be identified with n pattern fragments, then we require

$$-n\frac{\Delta S}{S} \approx -ln\ P_n \approx 0.7.$$

Suppose that the totality of patterns is of the order of thousands of average sized identifier patterns, then $\frac{\Delta S}{S} \approx 0.001$ and $n \approx 7000$. This means that we would need several thousand tests per suspicious object in order to obtain a fifty percent chance of identifying it as malignant. Obviously this is a very large number, and it is derived using a standard argument for biological immune systems [51], but the estimate is too simplistic. Testing code at random places in random ways is hardly efficient, and while it might work with huge numbers in a three dimensional environment in the body, it is not likely to be a useful idea in the one-dimensional world of computer memory. Computers cannot play the numbers game with the same odds as biological systems. Even the smallest functioning immune system (in young tadpoles) consists of 10^6 lymphocytes, which is several orders of magnitude greater than any computer system.

What one lacks in numbers must therefore be made up in specificity or intelligence. The search problem is made more efficient by making identifications at many scales. Indeed, even in the body, proteins are complicated folded structures with a hierarchy of folds which exhibit a structure at several different scales. These make a lock and key fit with receptors which amount to keys with sub-keys and sub-sub-keys and so on. By breaking up a program structurally over the scale of procedure calls, loops and high level statements one stands a much greater chance of finding a pattern combination which signals danger. Optimally, one should have a compiler standard to facilitate this. The executable format of a program might reveal weaknesses. Programs which do stack long-jumping or use functions gets() and scanf() are dangerous, they suggest buffer overflows and so forth. It is possible that systems could enforce obligatory segmentation management on such programs, with library hooks such as Electric Fence [60]. Unfortunately such hooks incur large performance overheads, but this could also be optimized if operating systems provided direct support for this.

Permanent programs should be screened for dangerous behavior once and for all, while more transitory user programs could be randomly tested. In this way we effectively distinguish between self and non-self, by adoption, for the sake of efficiency. There is no reason to go on testing system programs provided there is adequate security. In periods of low activity, the system would use its inactivity to make spot checks. The most adaptable strategy would be to leave a hook in each application or service (sendmail, ftp, cfengine) which would allow a subroutine antibody to attach itself to the program, testing the system state during the course of the program's execution. Problems would then be communicated back to the system.

Another possibility is that programs would have to obey certain structural protocols which guaranteed their safety. Graham et al have introduced the notion of adaptable binary programs [61]. This is a data format for compiled programs which allows adaptable relocation of code and analysis of binary performance without re-compilation. The ability to measure information about the performance and behavior of executable binaries has exciting possibilities for security and stability, but it also opens programs to a whole new series of viral attacks which might hook themselves into the file protocol.

The biological danger model also suggests mechanisms here. It purports that a cell which dies badly signals danger. The analogue in program execution is that programs which do not end with a SIGCHILD (normal programmed death) but with SIGABRT, SIGBUS or SIGSEGV etc are dangerous; see Figure 2. If the system kernel could collect statistics about programs which died badly, it would be possible to warn about the need to secure a replacement (transplant) for a key program or to restart essential services, or even to purge the program altogether.

Signal	Cause
SIGINT	Interrupt/break or CTRL-C
SIGTERM	Terminate signal
SIGKILL	Instant death
SIGSEGV	Segmentation (memory) violation
SIGBUS	Bus error/hardware fault
SIGABRT	Abnormal termination
SIGILL	Illegal instruction
SIGIOT	Hardware fault
SIGTRAP	Hardware fault
SIGEMT	Hardware fault
SIGCHLD	Child process exiting (apoptosis)

Figure 2: Some common signals from signals.h .

In the long run, it will be necessary to collect more long term information about the system. Biological systems do this by Darwinism, by playing the game of huge numbers. Computers will have to be more refined than this.

More Feedback Systems and Reactors

Feedback in system administrations leads to some ipowerfuldeas. Computer systems driven by economic principles can provide us with a model of coping with excess load. The Market Net project [62] is developing technologies based on the notions of a market economy. This includes protocols and algorithms which adapt to changing resource availability. Resources, including CPU time, storage, sensors and I/O bandwidth, can be traded. When resources become scarce, prices rise (i.e., priorities wane) encouraging clients to adapt their resource usage. Such a system could come under attack through fraud. A consumer of services could make deceive the resource disseminator in an attempt to divert the system's wealth. Mechanisms must be in place to recognize this kind of fraud and respond to it to prevent exploitation of the systems. The kernel, as resource manager, needs to be aware of how many clones of a particular process or thread are active, for instance, and be able to restrict the numbers so as to preserve the integrity of the system. Fixed limits might be appropriate in some cases, but clearly the performance of the system could be optimized in some sense using a feedback mechanism to regulate activity. Biological and social systems adapt in just this way and a computer immune system should be able to adapt using a mechanism of this type.

The economy model holds some obvious truth, but the analogy is not quite the right one. It misses an important point: namely that operating system survival depends not only on the fair allocation of resources, but also on the ability to collect and clean up its waste products: the fight against entropy. Natural selection (evolution) is the mechanism which extends market philosophy to the real world. It includes not just resource sharing but also the ability to mobilize antibodies and macrophages which can actively redress imbalances in system operation.

From a physicists perspective a computer is an open system: a non-equilibrium statistical system. One can expect to learn from the field of statistical physics [63], field theory [64] and neural networks [65] as can biological studies.

Protocols

Protocol solutions are common in operating systems for a wide range of communication scenarios: there is security in formality. Protocols make the business of verifying general transactions easy. When it comes down to it, most operations can be thought of as transactions and formalized by procedural rules. The advantage of a protocol is the additional control it offers; the disadvantage is the overhead it entails. It is not difficult to dream up protocols which provide assurances that system integrity is not sacrificed by individual operations. Protocol solutions for system well-being could likely solve problem, in principle, but the cost in terms of overhead would not be acceptable. A balance must be stricken whereby basic (atomic) system transactions are secured by efficient protocols and are supplemented by checks after the fact. Still, as computing power increases, it becomes viable (and for some desirable) to increase the level of checking during the transactions themselves. Let us mention a few areas where protocol solutions could assist computer immunity.

i) Process dispatch, services and the acceptance of executable binaries from outside. Programs could be examined, analyzed and verified before being accepted by the system for execution, as with the Java Virtual Machine. Hostile programs could be marked hostile with 'antibodies' and held inert, while safe programs could be marked safe with a public key. Spot checks on existing safe programs could be made to verify their integrity, perhaps using checksums, such as md5 checksums. ii) Object inheritance with histories: program X can only be started by a named list of other programs. This is like TCP wrappers/rsmsh but within the confines of each host. A linking format allows us to place hooks in a program to which the OS can attach test programs, a bit like a debugger. In this way, one could perform spot checks at run time from within. This also opens a new vulnerability to attack, unless one restricts hooks to the system. iii) License server technology is an example of software which will only run on a given host. Could one prevent people from sending native code programs to remote systems in this way? The Internet worm only propagated between systems with binary compatibility. iv) Can we detect when a program will do harm? One could audit system calls made by the program before running it in privileged mode. Detecting buffer overflows is one of the most important problems in present day computing. Electric fence etc. Of course this kind of computer system bureaucracy will slow down systems. v) Spamming could be handled by equipping reactors with a certain dead time, as one finds in neuronic activity. Adaptive locks [37] solve this problem. They could be used to limit the availability of critical and non-critical services in different ways. For example, after each ping transaction, the system would not respond to another ping transaction for a period of *t* seconds.

Each of these measures makes our instantaneous computer systems closer to sluggish biological systems, so it is important to choose carefully which services should be limited in this way.

Learning Systems

Seemingly inert molecular systems have a memory of previously fought infectious agents. This is not memory in the sense of computers but a memory in the Darwinian sense formed by the continual reappraisal of the system's sense of priorities. Computers cannot work in this way: the number of players in computer systems is many orders of magnitude too small. What they can do however is to learn from past experience.

Time series prediction is a way of predicting future behavior based on past experience. Watching logs and process signals, we can build up a pattern of activity and use it to sense difficulty. Time series detection is well established in seismology, vulcanology and astronomical observation. The only difference here is that the data form a discrete alphabet of events rather than continuous measurements. Patterns need to be established: looking for regularly occurring problems such as lack of memory or swapping/paging (thrashing) fits, which disks become full, as well as process sequences which most often lead to difficulty. Advanced state detection can recognize symptoms before they develop into a problem. Fuzzy 'logic' and behavioral pattern recognition are natural ways to diagnose developing situations such as disk-full conditions and attacks to the system. Pattern recognition and neural networks will be useful for diagnosing external attacks on the system as well as for diagnosing cases where the system attacks itself.

Logging probes like Network Flight Recorder and Bro [25, 24] can be used to collect the information, but a proper machine analysis of the data is required. System logs also need to be analyzed: can we reduce complex log messages to strings of simple characters [26] ? What is the alphabet of such messages? What is the scale of the signals? At the small scale (lots of detail) we have network protocols. At the large scale (averaged changes over long times) we have statistical entropy and load patterns other measures.

Information, Time Series and Statistical Mechanics

A multitasking computer, even a stand-alone computer, is a complex system; coupled to a network, its level of complexity increases manifold. Although scarcely reaching the level of biological or social complexity, computer networks could provide us with an ideal testing ground for many issues in those fields at the same time as being worthy of study for purely practical reasons. Complex systems have been analyzed in the context of physics and biology. The methodology is well known to experts, if not completely understood. Future computer systems will benefit from the methods for unravelling complexity as the level of distribution and cooperation increases. In many ways this harks back to Asimov's psycho-history: the ability to predict social trends based on previous behavior.

Complexity in a computer system arises both from the many processes which are running in the kernel and from the distribution of data in storage. System activity is influenced by the behavior of users. Users exert a random influence on the system leading to fluctuating levels of demand and supply for resources. Overlaid across this tapestry of fluctuating behavior we can also expect some strong regular signals. We expect to find a number of important regularities: daily, hourly, and weekly patterns are to be expected since these are the frequencies with which the most common cron jobs are scheduled. They also correspond to the key social patterns of work and leisure amongst the users of the system. All students rush to the terminal room at lunch time to surf the web; all company employees run from the terminal room at lunch time to sit in the sun. The daily signal will perhaps be the strongest since most humans and machines have a strong daily cycle.

Home grown periodic behavior is easily dealt with: if we expect it, it does not need to be analyzed in depth. However, other periodic signals might reflect regular activity in the environment (the Internet for instance) over which we have no control. They would include everything from DNS domain transfers to programmed port scanning. They affect our own systems, in perhaps subtle but nonetheless important ways which reflect both the way in which network resources are shared between uncooperating parties and the habits of external users seeking their gratification from our network services. Periodic patterns can be discovered in a variety of ways: by Fourier analysis and by search algorithms, for instance. A further possibility which has important potential for the general problem of behavior analysis is the use of neural networks. Neural networks lead us into the general problem.

In a complex system, it is not practical to keep track of every transaction which occurs, nor is it interesting to do so. Many events which take place cause no major changes in the system; there are processes constantly taking place, but their effect on average is merely to maintain the status quo. In physics one would call this a dynamical equilibrium and random incoherent events would be called noise. Noise is not interesting, but a clear signal or change in the system average behavior is interesting. We are interested in following these major changes in computer systems since they tell us the overall change in the behavior with time; see Figure 3. On a stable system we would not expect the average behavior of the system to change very much. On an unstable system, we would expect large changes.

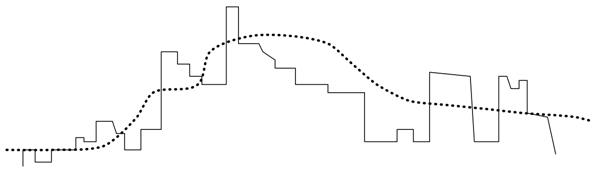

Figure 3: Although the details of system behavior seem random, the averages can reveal trends which are simpler to deal with.

The implication in the preceding sentence is based on the prejudice that significant change is a bad thing. That point of view might be criticized. What makes the gist of the argument correct is that it is always possible to define a measured quantity in such as way that this is true. A certain level of chaos might be acceptable or even desirable, according to one definition of chaos, but unacceptable according to another. In other words, the formulation of the problem is central. The identification of the correct metrics is a subject for future research, probably more lengthy and involved than one might think.

There is a close analogy here with the physics of complex systems. At the simplest level the equilibrium state of a system and its average load has a thermodynamical analogy: namely in terms of quantities analogous to temperature, pressure and entropy. If one imagines defining a system's average temperature and pressure from the measured averages of system activity, then it is reasonable that these will follow a normal thermodynamic development over long times. From a physical point of view, a computer shares many features in common with standard thermodynamical models. The idea of using average parameters to characterize the behavior is similar to what programs such as `xload` or Sun's `perfmeter` do. There are also other ways [37] in which to record the local history of the system. To put it flippantly we are interested in computer weather forecasting. But there there is much more to be gained from the computation of averages than plotting line graphs to inform humans about the recent past. The ability to identify trends and patterns in behavior can allow a suitably trained autonomous system to take measures to prevent dangerous situations from occurring before they become so serious that it becomes necessary to fetch a 'doctor.' The reason why single messages are insufficient is that computer systems are clearly to a large extent at the mercy of users' behavior. If one understands local habits and work patterns, then preventative action can be diagnosed and administered without having to rely on the immediate availability of humans doctors and technicians. Long term patterns cannot necessarily be understood from singular log messages or threshold values of system resources. There are too many factors involved. One must instead grasp the social aspect of system usage in an approximate way.

It is interesting to remark that, by averaging over the discrete behavior of a complex system, one can end up with continuously varying potentials; see Figure 4. Possibly computer networks will at some stage of the future be reinterpreted as analogous electric circuits in which the potentials are not electricity but statistical events characterizing the flow of activity throughout. Simple conservation arguments should be enough to convince anyone that what one ends up with is simply the physics of an abstract world forged by the imprint of information flows. Much of this is implicit in Shannon's original work on information theory [66]. It should be emphasized that the physics of complex cooperative systems is one of the most difficult unsolved problems of our time so quick answers can easily be discounted. Nonetheless, there is cause for optimism: often complexity is the result of simple transactions and simple mechanisms. My guess is that, to a useful level of approximation, the analysis of computer systems will prove to be relatively straightforward, using the physics of today, just as biological studies are benefitting from such theoretical ideas.

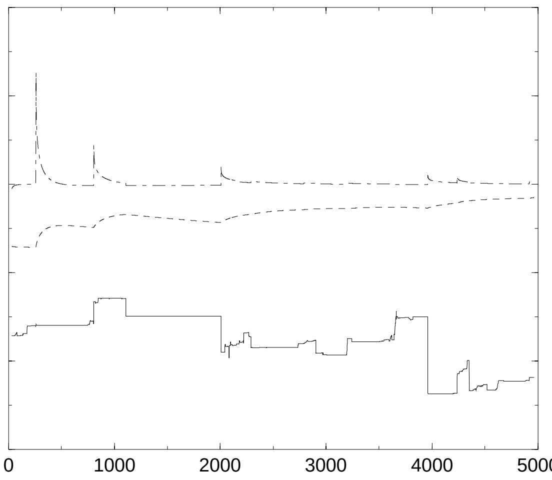

Figure 4: Disk usage as a function of time over the course of a week, beginning with Saturday. The lower solid line shows actual disk usage. The middle line shows the calculated entropy of the activity and the top line shows the entropy gradient. Since only relative magnitudes are of interest, the vertical 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.

Summary: Putting the Pieces Together

All of the ideas noted in this paper have been discussed previously in unrelated academic contexts. The expertise required to build a computer immune system exists in fragmented form. What is now required is a measure of imagination and a considerable amount of experimentation in order to identify useful mechanisms put together the pieces into a working model. Fortunately there is no shortage of ingenuity and willingness to participate in this kind of experimentation in the system administration community.

The best immune system one could build today would be made up the elements such as those in Table 1.

With these tools, each host is as self-contained as possible, accepting as little outside data as can be. Sharing of Bro/Network Flight Recorder data should be done carefully to avoid it being used as a means of manipulating the system. In the absence of a better running analysis, it is difficult to do better than this. Even so, with carefully thought out rules, this provisional approach can be very successful. Unfortunately, finding the best rules is presently a time-consuming job for an experienced system administrator. In time, perhaps we shall assemble a generic database of rules for cfengine and related tools.

Hopefully a computer immune system will at some time in the future become a standard. The last thing we need is a multitude of incompatible systems from a multitude of vendors. Free software such as GNU/Linux could blaze this trail since it is open for development and modification in all its aspects and could prevent important mechanisms from being patented. Few vendors are quick to adopt new technology, but one might hope that a properly designed fault preventive system would be more than they could resist. A POSIX standard which laid the groundwork for computer immunology is something to aim for. Future papers on this subject must lay down the operating system requirements for this to happen.

Am I trying to send the message that system administration is a pointless career, an inferior pursuit? No, of course not. An immune system cannot no more replace the system administrator than a lymphocyte can replace a surgeon, but an immune system makes the surgeon's existence bearable, fighting the stuff that is not easy to see and requiring basically no intelligence. Many of the ideas in this paper have an artificial intelligence flavor to them, but the main point is that immune systems in nature are far from

Convergence	cfengine	*Build expert systems for configuration Correlate system state and activity using switches or 'classes' to activate responses.*
Detection	Bro/N.F.R.	*Intrusion detection based on event occurrence. simple analysis switches on classes in* cfengine *with predetermined rules to counter intrusion attempts.*
	load average	*Process load average used to detect thresholds which switch feed back class data to cfengine. Cfengine restricts access to services, kills offending processes etc.*

Table 1: A makeshift immune system today. The key points this addresses are convergence and adaptive behavior.

intelligent. The less intelligent our autonomic systems are the cheaper they will be. Nature shows us that responsive system's don't need much intelligence as long as their mechanisms are ingenious! Simplicity and frequency are the keywords. I hope that the next few years will see important advances in the development of cooperative systems with the task of preserving the general health and reliability of the network.

I am grateful to Ketil Danielsen for a discussion about market economies in computing.

Note Added

After completing this paper, I was made aware of reference [67] where the authors conduct a time-series analysis of Unix systems very similar to those which I have advocated here. This paper deserves much more attention than I have been able to give it before the submission deadline.

Author Information

Mark Burgess is associate professor of physics and computer science at Oslo College. He is the author of GNU cfengine and can be reached at http://www.iu.hioslo.no/˜mark, where you will also find all the relevant information about cfengine and computer immunology.

Bibliography

[1] John Brunner. *The Shockwave Rider*. Del Rey, New York, 1975.

[2] M. W. Eichin and J. A. Rochlis. "With microscope and tweezer: an analysis of the internet worm." *Proceedings of 1989 IEEE Computer Society symposium on security and privacy*, page 326, 1989.

[3] M. Burgess. "A site configuration engine." *Computing systems*, 8:309, 1995.

[4] M. Burgess and R. Ralston. "Distributed resource administration using cfengine." *Software practice and experience*, 27:1083, 1997.

[5] M. Burgess. "Cfengine as a component of computer immune-systems." *Norsk informatikk konferanse*, page (submitted), 1998.

[6] M. Burgess and D. Skipitaris. "Acl management using gnu cfengine." *USENIX ;login:*, Vol.23 No. 3, 1998.

[7] Isaac Asimov. *I, Robot*. 1950.

[8] P. Anderson. "Towards a high level machine configuration system." *Proceedings of the 8th Systems Administration conference (LISA)*, 1994.

[9] M. Fisk. "Automating the administration of heterogeneous LANs." "Proceedings of the 10th Systems Administration conference (LISA)," 1996.

[10] J. P. Rouillard and R. B. Martin. "Config: a mechanism for installing and tracking system configurations." *Proceedings of the 8th Systems Administration conference (LISA),* 1994.

[11] J. Finke. "Automation of site configuration management." *Proceedings of the 11th Systems Administration conference (LISA)*, page 155, 1997.

[12] M. Burgess and D. Skipitaris. "Adaptive locks for frequently scheduled tasks with unpredictable runtimes." *Proceedings of the 11th Systems Administration conference (LISA)*, page 113, 1997.

[13] R. Evard. "An analysis of unix system configuration." *Proceedings of the 11th Systems Administration conference (LISA)*, page 179, 1997.

[14] D. Schmidt. "Ace. adaptive communication environment." http://http://siesta.cs.wustl.edu/schmidt/ACE.html.

[15] Tivoli systems/IBM. *Tivoli software products*. http://www.tivoli.com .

[16] Hewlett Packard. *Openview.*

[17] Sun Microsystems. *Solstice system documentation*. http://www.sun.com .

[18] Host factory. *Software system.* http://www.wv.com.

[19] W. F. Tichy. "RCS – A system for version control." *Software practice and experience*, 15:637, 1985.

[20] Caldera. *COAS project*. http://www.caldera.com .

[21] *Webmin project.* http://www.webmin.com .

[22] Palantir. The palantir was a project run by the university of Oslo Centre for Information Technology (USIT). Details can be obtained from palantir@usit.uio.no and http://www.palantir.uio.no . I am informed that this project is now terminated.

[23] S. E. Hansen and E. T. "Atkins. Automated system monitoring and notification with Swatch." *Proceedings of the 7th Systems Administration conference (LISA)*, 1993.

[24] V. Paxson. "Bro: A system for detecting network intruders in real time." *Proceedings of the 7th USENIX security symposium*, 1998.

[25] M. J. Ranum, et al. "Implementing a generalized tool for network monitoring." *Proceedings of the 11th Systems Administration conference (LISA)*, page 1, 1997.

[26] R. Emmaus, T. V. Erlandsen, and G. J. Kristiansen. *Network log analysis*. Oslo College dissertation., Oslo, 1998.

[27] S. Kittur, et al. "Fault tolerance in a distributed chorus/mix system." *Proceedings of the USENIX Technical conference.*, page 219, 1996.

[28] W. Rosenbery, D. Kenney, and G. Fisher. *Understanding DCE.* O'Reilley and Assoc., California, 1992.

[29] J. S. Plank. "A tutorial on Reed-solomon coding for fault tolerance in RAID-like systems." 27:995, 1997.

[30] R. Pike, D. Presotto, S. Dorwood, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. "Plan 9 from Bell Labs." *Computing systems*, 8:221, 1995.

[31] S. J. Mullender, G. Van Rossum, A. S. Tannenbaum, R. Van Renesse, and H. Van Staveren. "Amoeba: a distributed operating system for the 1990s." *IEEE Computer*, 23:44, 1990.

[32] A. S. Tannenbaum, R. Van Renesse, H. Van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, and G. Van Rossum. "Experiences with the amoeba distributed operating system." *Communications of the ACM*, 33:46, 1990.

[33] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. "The design and implementation of arjuna." *Computing systems*, 8:255, 1995.

[34] The Object Management Group. "Corba 2.0, interoperability: Universal networked objects." *OMG TC Document 95-3-10*, Framingham, MA, March 20, 1995.

[35] H. Wasserman and M. Blum. "Software reliability via run-time result-checking." *J. ACM*, 44:826, 1997.

[36] Mark Burgess. *GNU cfengine*. Free Software Foundation, Boston, Massachusetts, 1994-1998.

[37] M. Burgess. "Automated system administration with feedback regulation." *Software practice and experience*, (To appear), 1998.

[38] M. Carney and B. Loe. "A comparison of methods for implementing adaptive security policies." *Proceedings of the 7th USENIX security conference*.

[39] N. Minsky and V. Ungureanu. "Unified support for heterogeneous security policies in distributed systems." *Proceedings of the 7th USENIX security conference*.

[40] SANS. "System administration and network security." http://www.sans.org .

[41] USENIX. "Operating system protection for fine-grained programs." *Proceedings of the 7th USENIX Security symposium*, page 143, 1998.

[42] I. S. Winkler and B. Dealy. "A case study in social engineering." *Proceedings of the 5th USENIX security symposium*, page 1, 1995.

[43] J. Su and J. D. Tygar. "Building blocks for atomicity in electronic commerce." *Proceedings of the 6th USENIX security symposium:97, 1996.*

[44] W. Venema. "Murphy's law and computer security." *Proceedings of the 6th USENIX security symposium*, page 187, 1996.

[45] F. M. Burnet. *The Clonal selection theory of acquired immunity*. Vanderbilt Univ. Press, Nashville TN, 1959.

[46] F. M. Burnet and F. Fenner. *The production of antibodies*. Macmillan, Melbourne/London, 1949.

[47] J. Lederberg. *Science*, 1649:129, 1959.

[48] R. E. Billingham, L. Brent, and P. B. "Medawar." *Nature*, 173:603, 1953.

[49] R. E. Billingham. *Proc. Roy. Soc. London.*, B173:44, 1956.

[50] P. Matzinger. "Tolerance, danger and the extended family." *Annu. Rev. Immun.*, 12:991, 1994.

[51] A. S. Perelson and G. Weisbuch. "Immunology for physicists." *Reviews of Modern Physics*, 69:1219, 1997.

[52] Linus Pauling and Dan H. Campbell. "The production of antibodies in vitro." *Science*, 95:440, 1942.

[53] G. Edelman. (unknown reference), later awarded Nobel prize for this work in 1972 with R. R. Porter, 1959.

[54] R. R. Porter. (unknown reference), later awarded Nobel prize for this work in 1972 with G. Edelman, 1959.

[55] I. Roitt. *Essential Immunology*. Blackwell Science, Oxford, 1997.

[56] A. C. Clarke and S. Kubrick. *2001: A space odyssey*. MGM, Polaris productions, 1968.

[57] I. Goldberg, et al. "A secure environment for untrusted helper applications." *Proceedings of the 6th USENIX security symposium.*, page 1, 1996.

[58] Sun Microsystems. *Java programming language.* http://java.sun.com/aboutJava/ .

[59] C. Cowan, et al. *Stackguard project.* http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/.

[60] Pixar, B. Perens. "Electric fence, malloc debugger." *Free software foundation*, 1995.

[61] S. Graham, S. Lucco, and R. Wahbe. "Adaptable binary programs." *Proceedings of the USENIX Technical conference.*, page 315, 1995.

[62] Market net project. *A survivable, market-based architecture for large-scale information systems.* http://www.cs.columbia.edu/dcc/MarketNet/ .

[63] F. Reif. *Fundamentals of statistical mechanics*. McGraw-Hill, Singapore, 1965.

[64] Mark Burgess. *Applied covariant field theory*. http://www.iu.hioslo.no/mark/physics/CFT.html, (book in preparation).

[65] J. A. Freeman and D. M. Skapura. *Neural networks: algorithms, applications and programming techniques*. Addison Wesley, Reading, 1991.

[66] C. E. Shannon and W. Weaver. *The mathematical theory of communication*. University of Illinois Press, Urbana, 1949.

[67] P. Hoogenboom and J. Lepreau. "Computer system performance problem detection using time series models." *Proceedings of the USENIX Technical Conference, Summer 1993*, page 15, 1993.