Emergent (Mis)behavior vs. Complex Software Systems

Jeffrey C. Mogul
HP Laboratories Palo Alto

HPL-2006-2
December 22, 2005*

emergent behavior, emergent misbehavior, complex systems, distributed systems

Complex systems often behave in unexpected ways that are not easily predictable from the behavior of their components; this is known as *emergent behavior*. As software systems grow in complexity, interconnectedness, and geographic distribution, we will increasingly face unwanted emergent behavior.

Unpredictable software systems are hard to debug and hard to manage. We need better tools and methods for anticipating, detecting, diagnosing, and ameliorating emergent misbehavior. These tools and methods will require research into the causes and nature of emergent misbehavior in software systems.

* Internal Accession Date Only

Approved for External Publication

Emergent (Mis)behavior vs. Complex Software Systems

Jeffrey C. Mogul
HP Labs, Palo Alto
Jeff.Mogul@hp.com

Abstract

Complex systems often behave in unexpected ways that are not easily predictable from the behavior of their components; this is known as *emergent behavior*. As software systems grow in complexity, interconnectedness, and geographic distribution, we will increasingly face unwanted emergent behavior.

Unpredictable software systems are hard to debug and hard to manage. We need better tools and methods for anticipating, detecting, diagnosing, and ameliorating emergent misbehavior. These tools and methods will require research into the causes and nature of emergent misbehavior in software systems.

1 Introduction

Most systems research papers describe new or better ways to do things. This should not be surprising; computer science is primarily an engineering science, not a natural science, and so our focus is usually on innovation, not on understanding the world as it is.

Some OS researchers have, however, looked at understanding and predicting system behavior, rather than designing and optimizing it. Why this shift in emphasis? One could argue that we have perhaps innovated too freely; the world seems not to urgently need another OS kernel, or another distributed shared memory protocol. And most of our optimizations are either too minor or too disruptive to influence widespread practice.

But another explanation for the shift lies in the complexity of the systems we build. The behavior of a simple system is often easy to understand as the sum of the behavior of its component parts; good engineering practice is to design components with well-defined and reliable behaviors for precisely this reason. As systems become more complex, this reductionist way of understanding them fails; they behave in ways that cannot feasibly be predicted from understanding of the individual parts, or were not expected by the system designer who assembled the parts, or both.

The term “emergent behavior” (or sometimes “emergence” or “ensemble behavior”) has been used to describe how complex behaviors arise out of simpler ones:

> *Emergent behavior is that which cannot be predicted through analysis at any level simpler than that of the system as a whole. Explanations of*

> *emergence, like simplifications of complexity, are inherently illusory and can only be achieved by sleight of hand. This does not mean that emergence is not real. Emergent behavior, by definition, is what's left after everything else has been explained.* – George Dyson [12, p. 9]

Dyson's definition is not the only one, and it oversimplifies; for example, how does one define the boundaries of "the system as a whole," when networks connect virtually all of our systems at some level? However, this definition captures the central concept.

Emergent behavior can be beneficial. (Individual ants are dumb; ant colonies are smarter.) But it is not always beneficial. For example, stock market panics are a form of unwanted emergent behavior in which the irrational behavior of many individual investors makes things worse for everyone. London's newish Millennium Footbridge had to be closed after "unexpected excessive lateral vibrations" on its opening day, which were due to an unexpected synchronization that built up between the footfalls of pedestrians and the motion of the bridge [11].

I will use the term "emergent misbehavior" [1] to focus on problematic behavior. I exclude the problem of intentionally malicious misbehavior from this definition; although attackers could exploit emergent behavior, that should be considered as a separate problem. This paper also will also avoid discussing situations involving game theory, in which multiple non-malicious actors are trying to exploit their knowledge of each other's behavior; this is a topic for future consideration.

Even when emergent behavior is not inherently bad, it is (by Dyson's definition) unpredictable, and unpredictability is bad in many computer system contexts – especially when it comes to performance. If one cannot predict the useful bandwidth of a network, or the number of transactions per second from a server, this makes it hard to design and manage computer systems. While emergent behavior is not the only cause of unpredictability, it is a central challenge for systems researchers. We are responsible for designing many of the mechanisms that maintain the performance "of the system as a whole," both on single computers and in distributed systems.

This paper will also argue that we need better tools and methods for anticipating, detecting, diagnosing, and ameliorating emergent behavior. Although Dyson's definition suggests that such tools and methods will always be imperfect, that should not stop us from trying.

2 Examples of emergent misbehavior

To motivate the rest of this paper, this section presents a few examples of emergent misbehavior, in both non-computer and computer systems. Other examples are scattered throughout the paper.

[1] The term has been used before by others; for example, Nisley [30].

2.1 Non-computer examples

On the first day that the Millennium Footbridge was opened to significant pedestrian traffic, “unexpected excessive lateral vibrations occurred,” causing “a significant number of pedestrians ... to have difficulty in walking.” [11] The bridge had be closed until the engineers analyzed and fixed the problem. The designers had failed to anticipate an effect that could cause the synchronization of individual footfalls, both with each other and with the bridge's natural swaying frequency: pedestrians on a swaying surface tend to synchronize their footsteps with the sway, even if the amplitude is initially quite small. The bridge would not have behaved in an unexpected way had not the pedestrians also shown unexpected behavior.

The Millennium Footbridge problem is somewhat surprising, since we expect bridge designers to understand this general kind of problem. In particular, the infamous failure of the Tacoma Narrows Bridge, four months after it opened in 1940, must surely be well known to every bridge designer in the world [36]. That bridge failed not because it was too heavy, but because in high winds its shape generated enough lift to induce major oscillations, and it was insufficiently resistant to torsional forces.

So even in a well-regulated engineering profession with decades or centuries of experience with unexpected dynamic failures, and with regular use of computer modelling, modern designs such as the Millennium Footbridge still suffer from emergent misbehavior. That should keep us humble.

The civil engineering literature shows an awareness of the possibility of emergent behavior on the part of people who use their systems. For example, “[automotive traffic] is emergent behavior, i.e. the result of the individual decisions of drivers, pedestrians, traffic controllers and other individuals.” [13]. Many traffic jams are emergent misbehavior; traffic slows or stops even when there is no inherent impediment to its flow.

2.2 Computer hardware examples

We tend to treat disk drives as components that interact with each other, if at all, through storage controllers and storage protocols such as SCSI. In large installations, however, large numbers of disk drives are mounted on racks. It turns out that the performance of a drive can be adversely affected by the vibrations caused by seek activity on neighboring drives [2]. Disk drive manufacturers have learned to engineer “enterprise” drives to resist this behavior, which is one reason why they cost more than consumer-market drives.

2.3 Examples from computer networking

The “Ethernet capture effect” is a clear case of emergent misbehavior [33]. The capture effect creates significant unfairness in certain CSMA/CD environments.

Consider the case of a short LAN with exactly two hosts A and B, both with a lot of packets to send, when a third host sends its last packet for a while. A and B will simultaneously detect that the channel is now free, both will send, and the collision will cause both to calculate a random backoff.

Suppose that A chooses a smaller backoff than B. Then A will send, after which both hosts again see the channel become free, and both send again, resulting in another collision. However, since A “won” the first collision, its collision counter has been reset, and the expected value of B's random backoff is larger than A's. So A will probably win again, and B's chances get progressively worse.

This problem was not seen until Ethernet hardware had been in significant commercial use for many years. It only appeared once Ethernet chips were fast enough to fully exploit the timing allowed by the specification. Thus, the capture effect appeared not because of a “problem” with any of the components, but because they were improved (in this local sense) to an optimal point. The solution was to require a host to insert a little extra delay if it might be the winning host in a capture-effect situation [33].

Routers periodically exchange routing protocol messages. One would hope that, in a large network, there is a fairly constant background level of routing protocol message traffic. However, Floyd and Jacobson showed that in a network with “many apparently-independent periodic processes ... these processes can inadvertently become synchronized.” [14] The transition is not gradual but abrupt, and is therefore hard to anticipate if one is not carefully looking for it.

Two hosts exchanging data using TCP can experience a bad interaction between the TCP sender's Nagle algorithm and the receiver's delayed-acknowledgment algorithm; the interaction is exacerbated by the traditional design of the network stack [28]. The problem is not just academic; users regularly encounter this, especially when using networks whose maximum packet size is larger than that of Ethernet.

2.4 Examples from distributed systems and operating systems

Figure 1 shows the structure of a simple multi-tiered distributed application, with numerous clients spread throughout the Internet, a front-end server, a load-balancer, two application servers, and two database servers. The overall application involves collecting periodic measurement reports from the clients, doing some processing at the application servers, and then storing the processed reports in the replicated database.

The load balancer in this system has two jobs: it spread the workload evenly among the available application servers, and it detects the failure of an application server and stops sending it work. The load balancer detects failure when an application server fails to respond to a request within a certain threshold latency (timeout).

Suppose that the system appears to be working perfectly when it is first put into service. However, as months go by, the database latency increases; perhaps the index efficiency gets worse as it gets larger, or perhaps the working set starts to exceed the size of the database's cache. Suppose also that the load balancer has been configured to use a relatively timeout for detecting that one of the application servers has failed.

At some point, the system stops responding to requests. The database latency has increased to the point where the application servers are no longer responding to the load

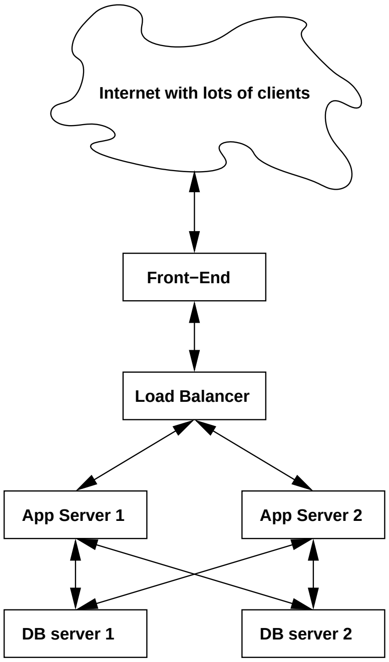

Figure 1: Example multi-tiered distributed system

balancer within its configured timeout, and the load balancer ends up declaring both application servers dead. In a variant of this problem, the load balancer alternates between application servers; as each server is forced to handle the full load, its latency exceeds the timeout, while the other, now unloaded server appears to recover.

Clearly the system as a whole is misbehaving. However, none of the components have failed, *per se*. One could argue that the database servers should not slow down over time, but this could be hard to guarantee. Or one could argue that someone chose the wrong timeout for the load balancer, but that error might not have been obvious when the system was first tested.

Note that this example is simplified and has been constructed from several real-world examples (which cannot be further described for reasons of confidentiality).

Readers should be able to see other well-known operating system problems, such as priority inversion and data races, as examples of emergent behavior.

2.5 Complex behavior in more complex systems

While the focus of this paper is on emergent misbehavior in complex systems, many of the examples show that emergent misbehavior can happen in extremely simple systems. For example, the Ethernet capture effect and the interaction between TCP's Nagle and delayed-ACK algorithms both can arise in two-computer systems with trivial applications. Perhaps the best example of a simple system exhibiting complex behavior is John

Conway's game of Life [16], in which three simple rules govern cellular automata on a grid. It is nearly impossible to predict the long-term behavior of even a small initial configuration in Life.

If we cannot avoid unexpected behavior in such simple systems, we are very unlikely to avoid it in complex computer systems. And while emergent behavior might be "fascinating" in the game of Life, it is usually undesirable in computer systems.

3 Related work

In 2001, Steven Gribble argued "against a seemingly common design paradigm that attempts to achieve robustness by predicting the conditions in which a system will operate, and then carefully architecting the system to operate well in those (and only those) conditions" [17]. Gribble's observations and conclusions overlap considerably with mine, but his proposed solutions focussed on "design strategies that help to make systems more robust in the face of the unexpected." My focus (in the following sections) is on gaining better understanding of emergent misbehavior in complex software systems, which I believe is a prerequisite for improved design strategies as well as improved system management tools and techniques.

Gribble specifically identified the problem of "unpredictable behavior in the face of small perturbation," or, more concisely, the "butterfly effect." (The term is generally ascribed to Edward Lorenz.) However, emergent behavior need not necessarily arise from a small perturbation; it might be inherent in the unperturbed behavior of the system as it is designed or implemented. For example, both of us refer to the example of livelock (see Section 5.1), but it is hard to see this as an example of the "butterfly effect" (although it does have a sudden onset).

Another way to look at this is that the butterfly effect applies to chaotic systems, where even if one can deduce the cause of one instance of misbehavior, one still has no more ability to predict the next instance. In many cases of emergent behavior (including most of the examples in Gribble's paper), it might well be possible to gain sufficient insight into a past unexpected behavior to be able to predict or prevent it in the future. (In fact, while many complex systems misbehave, it is hard to argue that all of these systems are actually chaotic.) Therefore, I believe the issue of "small perturbations" is a red herring, leading to excessive pessimism.

Others have certainly looked at the issue of emergent behavior in enterprise systems. For example, the emphasis on self-management in IBM's autonomic computing vision clearly leads to emergent behavior, as pointed out by Kephart and Chess [22], although they focussed more on how to encourage ("design") emergent *good* behavior, rather than to detect, diagnose, or prevent emergent misbehavior.

4 What is/is not emergent misbehavior?

A concept such as “emergent misbehavior” runs the risk of being applied both too broadly (“everything can be seen as emergent behavior”) and too narrowly (“that's not emergent behavior, because I can explain it as a simple, deterministic process”). So in order for it to be useful, we need some test for what constitutes emergent misbehavior and what does not.

Given the definition of emergent behavior as that “cannot be predicted through analysis at any level simpler than that of the system as a whole,” we can easily describe certain kinds of misbehavior that are clearly *not* emergent:

- **Single-component bugs that break the whole system:** If a critical component of the system simply stops working, one expects the system to fail, unless it is designed to survive component failure.
- **Inherently inefficient algorithms:** Some algorithmic choices are predictably inefficient. For example, a replicated file system that contacts replicas serially rather than in parallel will likely have sub-optimal performance. One could make this prediction without knowing how the replicas behave.
- **Insufficient resources:** The primitive resources (e.g., CPU, memory, network latency and bandwidth, storage capacity, latency, and bandwidth) inherently prevent the system from performing at the required level. For example, you cannot send a gigabyte of data over a 56 Kbit/sec dialup in 1 minute.

While Dyson wrote that “emergent behavior, by definition, is what's left after everything else has been explained,” it seems unsatisfactory to define emergent misbehavior simply as that which does not fit into one of the categories of predictable misbehavior. Approaching the question from the other direction, we can try to list properties common to some or all instances of emergent misbehavior[2]:

1. Inherently hard-to-predict behavior: Even when the rules governing a system's behavior are fully known and deterministic, it can be hard to predict how it behaves as a whole; if the system also includes probabilistic or non-linear components, or its scale is quite large, the prediction problem becomes much harder.
2. Sudden changes in behavior: If a system's behavior can change rapidly between modes with greatly different performance characteristics, its behavior will be hard to predict when the parameters that control this mode switch are near their critical point. For example, the Ethernet capture effect arose “suddenly” when chip designers managed to reduce the inter-packet gap to the minimum allowed by the specification.
3. Amplification of seemingly minor behaviors: Prediction is easier when we can ignore minor deviations from expected behaviors, especially in larger-scale systems where we hope these individual deviations are swamped by the law of large numbers. If, however, these minor deviations can be amplified through effects such

[2] Steven Gribble suggested this approach [18]

as resonances or coincidences, they can lead to unpredictable behavior unless the amplification mechanisms are understood.

One hard-to-resolve question is whether *chaotic* misbehavior is best understood as emergent or not. If one defines a chaotic system as one whose global behavior, while deterministic, is so sensitive to initial conditions that it appears to be unpredictable, then it has some of the same characteristics as emergence. This might seem to clash with Dyson's definition, which suggests that emergent behavior is ultimately predictable if understood at the right scale. However, another definition of emergence:

> *I'll not call a phenomenon* emergent *unless it is recognizable and recurring: when this is the case, I'll say the phenomenon is* regular. *That a phenomenon is regular does* not *mean that it is easy to recognize or explain.*
> – John Holland [20]

does seem to include chaotic behavior, if it is recurring. Also, Parunak and VanderBok [31] explicitly separate "emergent chaos" from true randomness.

4.1 Inductive vs. deductive understanding

Perhaps a convenient way to draw the line is to look at the best approach available to understanding the connection between system design and system performance. If one can start with a description of a system's components and configuration and reason forward (inductively) to accurately predict the system's behavior, then this behavior is not emergent.

If one cannot use induction, one might still be able to work from observations of the system's behavior and reason in reverse (deductively) to infer what actually happened, once something does go wrong. These systems exhibit emergent behavior, and given the deductive inference, there might be some hope of directly removing the causes of their emergent misbehaviors.

Finally, there are systems (for example, chaotic systems) where even in principle it is impossible to explain either inductively or deductively exactly what causes misbehavior. These systems also exhibit emergent behavior, but attempts to fix their emergent misbehaviors might be limited to finding ways to constrain the system or to nudge it out of misbehavior, rather than directly fixing the underlying cause.

Deductively-understandable emergent misbehavior is the most interesting kind, because once we understand what causes a particular misbehavior, we can usually fix it.

5 A research agenda

The main point of this paper is to propose a research agenda to deal with emergent misbehavior in complex software systems, with an initial focus on the operating system aspects of the problem.

This agenda parallels one that has been proposed in the context of distributed control systems for manufacturing systems, in a paper by Parunak and VanderBok [31]. Their paper described examples of emergent misbehavior in a specific domain, the use of automated welding systems. Several of the ideas presented here are based on their paper, somewhat transposed for the different problem domain.

Major issues on the proposed agenda include:

1. **Creating a taxonomy of emergent misbehavior**: What general kinds of emergent misbehavior do we see in software systems? Experience suggests that many, if not most, kinds of emergent misbehavior could indeed be put into a reasonably small number of categories, although it is not yet clear whether there is a large set of possible idiosyncratic emergent misbehaviors that are hard to categorize.
2. **Creating a taxonomy of typical causes**: We also need a taxonomy of frequent causes of emergent misbehavior, tied to specific instances of the taxonomy of misbehaviors.
3. **Developing detection and diagnosis techniques**: Given that emergent misbehavior is, almost by definition, unexpected, perhaps the key step in dealing with it is to detect it. We should develop techniques both to detect generic kinds of misbehavior, based on the taxonomy of misbehaviors, and to allow programmers and system maintainers to look for application-specific misbehaviors.
4. **Develop prediction techniques**: Even if emergent misbehavior is inherently hard to predict from first principles, that should not keep us developing techniques to predict it whenever possible, perhaps from advance symptoms.
5. **Develop amelioration techniques**: While it might be impossible to fully eliminate emergent misbehavior, it is certainly possible to reduce the chances that it will occur, both through careful system design and through generic techniques that address well-known causes.
6. **Develop testing techniques**: While improved detection mechanisms are useful in debugging an undeployed application and in monitoring a deployed application, most significant systems go through a testing phase between debugging and deployment. One goal of testing is to expose problems sooner than they would appear in real-life use; we need techniques to probe for plausible emergent misbehaviors during testing.

Each of these steps is covered in more detail below.

5.1 Create an emergent misbehavior taxonomy

As a first step, we need to understand general categories of emergent misbehavior in software systems. The list could include:

- **Thrashing**: Competition over a multiplexed scarce resource in which the costs of switching between the sharing parties dominates the useful work that can be performed. It is useful to distinguish this form of thrashing, which could be avoided through better scheduling or coordination, from unavoidable cases where a system is simply underprovisioned for the task at hand.

- **Unwanted synchronization**: A set of systems whose time-varying behavior should be uncorrelated instead ends up correlated. This means that resource allocations based on statistical multiplexing can fail. (The routing-message synchronization described by Floyd and Jacobson [14] and the Millennium Footbridge problem [11] both into this category, as does the possibly apocryphal story of municipal water systems failing when too many people flush their toilets during a commercial break in a popular TV program.)
- **Unwanted oscillation or periodicity**: A system oscillates between states because of an accidental or poorly-designed feedback loop between multiple components. Parunak and VanderBok describe an example from a collection of spot-welding robots [31].
- **Deadlock**: Progress stalls because of a circular set of dependencies. Deadlock can clearly result in a system where each component functions "correctly" except with respect to an arbitrary protocol for avoiding deadlocks, and they only result from the interactions between components.
- **Livelock**: The throughput of a system decreases, perhaps to zero, as the input rate increases past a certain point. Livelock differs from deadlock in that throughput is restored if the input rate decreases. Livelock can result when a system with multiple components, each of which is necessary for the complete processing of a request, gives too much priority to one of the components and hences starves another component as the system becomes saturated [29].
- **Phase change**: The behavior of a system changes radically as the result of an incremental change in some variable. In other fields, such as physics, such sudden changes can often be modelled as phase changes. Some computer systems can also exhibit phase change. For example, ad-hoc wireless networks often have critical thresholds, for local parameters such as per-node power levels, that control certain global properties, such as whether the network is mostly-connected or mostly-disconnected [24].

  As we develop systems of large numbers of relatively simple nodes (such as DHTs, sensor networks, in addition to ad-hoc wireless networks) where the nodes interact with each other, rather than with a global coordinator (e.g., a Web server), we might see additional examples of critical thresholds and densities that lead to phase changes.
- **Chaotic behavior**: as discussed in Section 4.

This taxonomy does not include "faults" or "component failures." In fact, none of the misbehavior examples in this paper stem from component failures. Their causes are inherent in the design or implementation of the system. Of course, a component failure could trigger a manifestation of system-level design failure.

It might be useful to arrange this taxonomy into a hierarchy. Parunak and VanderBok categorize three kinds of emergent behavior [31]:

1. Systems attracted to a fixed, stable point (perhaps not the desired operating point)

2. Oscillation
3. Chaotic operation

but these meta-categories might be specific to control systems, and insufficient for complex distributed systems. For example, we could add “sudden changes in behavior.”

5.2 Create a taxonomy of causes

Recognizing an instance of emergent misbehavior as a member of one of the categories listed above is simply a first step towards solving the problem. We also need a taxonomy of frequent causes of emergent misbehavior, tied to specific instances of the taxonomy of misbehaviors.

For example, the ultimate cause of a thrashing problem might be as simple as a memory leak (causing a program's address space to grow in a way that destroys locality). It might be failure to perform admission control, allowing too many otherwise well-behaved jobs into a system with limited resources. It might be an implementation bug in a scheduling algorithm, which in attempting to avoid poor scheduling decisions does exactly the opposite.

For each category from Section 5.1, it should be possible to build up a list of commonly-occurring generic causes. That list then can be applied in the search for the cause(s) of a specific emergent misbehavior problem. Such a list probably cannot be exhaustive – many systems might exhibit *sui generis* emergent misbehavior – but it could still cover a considerable number of cases.

Note that because emergent misbehavior is an aspect of an entire system, not of just one component, many or most of the causes in this list will themselves involve multiple components. For example, the cause of the receive livelock problem in an interrupt-driven network stack [29] was traced to the use of multiple, finite-length queues in the network protocol stack, along with the decision to give processing priority to the wrong queue.

In the context of control systems, Parunak and VanderBok state that nonlinearity causes emergent behavior, and that “[three] of the most common sources of nonlinearity are capacity limits, feedback loops, and temporal delays” [31]. All of these causes apply to software systems more generally, but there are other causes of emergent misbehavior, such as:

- **Unexpected resource sharing:** The system designer assumed that separate components had access to separate resources, when in fact the resources are shared and insufficient.
- **Massive scale:** The number of communicating components in the system is large enough to give rise to complex global behavior, even if individual components have simple behaviors.
- **Decentralized control:** We generally value decentralized system designs over centralized ones, even as we recognize that centralization often makes it easier to implement and manage a system. Huberman and Hogg [21] have provided a theoretical analysis of how distributed systems that lack central controls, and hence

suffer from incomplete knowledge and delayed information, can exhibit oscillations and chaos.

- **Unexpected inputs or loads:** Many systems react badly to unexpected inputs [27] or unexpected loads [7]. Not all such misbehavior is emergent. Remember, however, the point raised in Section 1 that it might be hard to define the boundaries of "the system as a whole," and sometimes implementors draw it too close to what they are responsible for implementing.

Both Parunak and VanderBok and Huberman and Hogg point out that delay is one of the principle contributors of emergent misbehavior. Delay is inherent in distributed and networked systems. While it might seem that the primary undesired consequence of latency is simply that the system will run slower, latency might be even more pernicious in how it makes a system harder to understand and harder to control. As Huberman and Hogg point out, delay (possibly aggravated by incomplete knowledge as the result of message loss) means that no single viewpoint can have a fully consistent and up-to-date view of global system state; this is what leads to oscillations and chaos.

Delay also creates emergent misbehavior for more mundane reasons. For example, system implementors often use timeouts to detect failure. Choosing the right timeout is seldom easy; static choices almost always fail sooner or later, and adaptive schemes are hard to design even for relatively simple cases such as TCP retransmissions.

5.3 Develop detection and diagnosis techniques

Given taxonomies of emergent misbehaviors and their causes, we can then develop techniques to detect emergent misbehavior, and perhaps even to diagnose their causes. In many cases, this might be the *best* that we can do, if emergent behavior is that which is inherently unpredictable.

To support detection, an operating system or distributed systems infrastructure could monitor its applications for generic patterns of behavior consistent with thrashing, livelock, unwanted periodicity, etc. This approach has shown success, with techniques such as the one described by Romer *et al.* for dynamic page mapping [35].

Parunak and VanderBok describe a number of general-purpose techniques for detecting emergent behavior in control systems, based on their division of causes [31]. For example, periodic behavior can be detected through Fourier analysis; similar techniques could be employed by operating systems and their associated management systems.

The diagnosis problem will be harder to solve. One approach might be to expose the system designer's expectations to the diagnosis system. Patrick Reynolds (with Janet Wiener, Amin Vahdat, myself, and several others) has developed a system, called Pip, for diagnosing behavior problems in distributed systems [34]. In the Pip approach, the programmer expresses expectations about system performance and causal structure, including both local and path-based global expectations. A middleware layer then monitors application behavior (including communication between nodes) to detect violated expectations. This approach builds on Perl and Weihl's "performance assertion checking" technique for parallel applications [32].

Note that the Pip approach does not depend on the ability of programmers to write formal (and correct) specifications, nor does it result in any proof of correctness. We expect programmers to initially create incorrect expectations, and then to evolve both these and the distributed system implementation, until the behavior seems correct and no violations remain. Pip does not attempt to eliminate the trial-and-error approach, only to make it less painful, and to gently force programmers to confront the possibility of unexpected behavior.

Systems designers can help the diagnosis effort by including enough monitoring and logging that diagnosis tools could construct a global view of system behavior, and at levels of detail so that unanticipated behavior can be captured. System designers tend to resist adding such “superfluous” monitoring because of its added runtime cost, but the costs of system failure can be even larger and certainly less predictable. (The Space Shuttle program provides an illuminating example: the Shuttle was deployed for two decades before NASA decided to use on-board cameras to see that foam was breaking off. [3]

Recent research aimed at the development of playback tools (for example, ReVirt [23]) and analysis tools (for example, Pinpoint [6], Cohen *et al.* [8], and Pip) increases the benefit of ubiquitous logging. Perhaps as the benefits become more broadly accepted as way to reduce the overall costs of managing complex systems, the minor operational costs of logging will become more acceptable.

5.4 Develop prediction techniques

In many contexts, it can be more important to have predictable performance than to have optimal performance. If performance is predictable but suboptimal, one can budget for the anticipated inefficiency (especially given that hardware costs are increasingly dominated by system administration costs). However, if performance is normally optimal but sometimes unpredictably bad, the system owner might be forced to plan for an arbitrary worst case. This, for example, would make it hard to set a competitive price for a service offering. Prediction therefore complements detection; presumably one would prefer to know about a potential problem in advance, not just after it has started.

Performance prediction covers many areas. For example, if there are no controls on the load imposed on the system (e.g., a Web server on the public Internet) then it might suffice to predict the patterns of load. But in many cases, the ability to predict emergent misbehavior could be quite useful.

The very concept of “predicting emergent behavior” might seem oxymoronic, given

[3] It is unclear whether the prior decision not to use cameras to look for foam problems was because NASA was trying to avoid the extra weight of 1980s-vintage cameras, or whether they were already in place but there was insufficient downlink bandwidth. The latter hypothesis is supported by a news report that the Columbia Accident Investigation Board recommended that that NASA “make the shuttle's on-board cameras, which capture images of the external tank after separation, available during the ascent, rather than just post-flight. That way, data may be used to assess debris strikes or other ascent anomalies earlier in the process.” [25].

Dyson's definition of emergent behavior as inherently unpredictable. This apparent paradox has two possible resolutions. First, Dyson's definition describes behavior "unpredictable through analysis at any level simpler than that of the system as a whole." This leaves open the possibility of prediction techniques that operate at the whole-system level. Second, while it might not be possible to predict *specific* emergent misbehavior, it might still be possible to predict that a system could be prone to *some* unspecified form of emergent misbehavior. Third, it might be possible to predict the onset of serious emergent misbehavior from advance symptoms.

Given that emergent misbehavior might often not be the result of component failure, traditional failure prediction techniques, such as those based on Mean Times Between Failures (MTBFs) or fault trees, might be inapplicable. MTBF data would only be useful if system-wide failures were primarily caused by component failures. One cannot build a fault tree that incorporates the probability of an unanticipated event. John Wilkes suggests, however, that it might be possible to work backwards from a bound on misbehavior, perhaps as imposed by a detection mechanism, to derive limits on the events that could provoke such misbehavior [37].

One possible approach to onset prediction would be the creation of a corpus of "signatures" based on observed events leading up to detected emergent misbehavior in real systems. When one or more such signatures are recognized in a running system, this could serve as an indicator that misbehavior is about to appear. For example, suppose one does a spectral analysis of response times at regular intervals. If the spectrum starts to include stronger frequency components than in the past, this could indicate the onset of oscillation before it becomes harmful.

The creation of such signatures could be guided by a taxonomy of causes, as described in Section 5.2. Of course, this approach cannot predict all misbehavior, and might not always generate predictions far in advance of real problems.

Cohen *et al.* [9] described a technique, based on statistical modelling and inference, that automatically extracts signatures from system metrics, especially during problem events. These signatures are constructed so that they can be matched against signatures for similar previous events; if the previous events are labelled with diagnoses, the matching events can suggest a diagnosis for a current problem. So far, they have only experimented with detection of component failure or overload, not with emergent misbehavior.

5.5 Develop amelioration techniques

In some cases, an emergent misbehavior might either be impossible to diagnose, or a valid diagnosis might point to a cause that cannot be fixed directly. In these cases, techniques for ameliorating or working around emergent misbehavior might be necessary.

For example, Floyd and Jacobson show how the injection of some extra randomness in the timing of routing updates can break up unwanted synchronization; they even "quantify how much randomization is necessary" [14]. Interestingly, Parunak and VanderBok also describe how randomization in timing can solve problems with defective welds from automated spot-welding guns [31].

Similarly, although it is possible in theory to modify a network stack to avoid livelock [29], in practice one might not have access to the source code. In this case, livelock can still be prevented by placing a rate-limiting box upstream from the system(s) subject to livelock; this box can discard excessive traffic soon enough that the remainder can be processed appropriately by the protected system.

Mary Baker [3] has pointed out that civil, structural, and mechanical engineers strive to avoid sudden failures. Their designs often sacrifice efficiency in favor of guaranteeing gradual failure, which gives time to react, and in favor of making it possible to regularly inspect for signs of impending failures. In an analogous distributed-system context, Maniatis *et al.* described how their peer-to-peer system explicitly uses rate-limiting “to prevent our adversary's unlimited resources from overwhelming the system quickly, and integrated intrusion detection to preempt unrecoverable failure” [26].

Gribble [17] suggested several design strategies, including the use of systematic over-provisioning, admission control, introspection, and closed control loops for adaptation. (However, experience such as reported by Parunak and VanderBok suggest that adding control loops might not solve the emergent behavior problem. Further, Brown and Hellerstein point out that adding automation, such as feedback control, to a simpler system can itself lead to unexpected behavior [4].) Gribble also suggested designing systems that expect failures and recover rapidly from them, rather than simply trying to design systems that never fail.

George Candea [5] points out that overall system dependability can be reduced by components that behave unpredictably, especially when buggy, stressed, or compromised. He suggests that system predictability can be improved by either by preventing unpredicted component behavior from propagating throughout the system, or by protecting components against unexpected inputs. For example, “software fuses” (such as firewalls) drop out-of-bounds inputs before they reach a vulnerable component; “output guards” detect apparent component failure and stop the suspect module, “thus coercing Byzantine into fail-stop behavior.” However, Candea's proposal assumes that misbehavior is apparent at either the input or the output of a component; system-wide (emergent) misbehavior might either be invisible at this level, or might be so pervasive that software fuses or guards would effectively shut down the entire system. A defense against emergent misbehavior is more likely to take the form of “damping” (to slow the propagation of problems) or “clamping” (to limit the amount of damage they can cause).

The goal of much distributed systems research has been the creation of complex systems that always work, both through fundamental design principles (e.g., two-phase commit and replication) and through better engineering (e.g., model checking and type-safe languages). However, the challenge of emergent misbehavior is that this “correct by construction” goal, while a worthy pursuit, probably will never be achieved, and we will always need amelioration techniques.

5.6 Develop testing techniques

No matter how good we are at developing techniques to avoid, diagnose, repair, and ameliorate emergent misbehavior, the complexity of any given situation could well confound these efforts. One might believe that an emergent misbehavior problem has been solved, when it has only been driven temporarily into hiding.

Therefore we will need techniques to test systems for emergent misbehavior. Testing for complex systems always poses challenges. For example, Armando Fox [15] suggests that the conditions that lead to emergent misbehavior are not always knowable or anticipated during testing.

A solution could include techniques for reproducing previously encountered emergent misbehavior, or rather the stimuli and configurations that led to them. It might also be possible to generate the conditions for emergent misbehavior automatically, based on the taxonomy of causes described in Section 5.2.

Other challenges include the need for automatic detection of emergent misbehavior (see Section 5.3), because extensive testing protocols must be automated and cannot rely on humans to detect if a test has failed.

6 Potholes on the roads to the future

Several computer companies have articulated ambitious visions for the future of complex computing systems, motivated by the increasing inability of unassisted humans to manage or comprehend these systems. These visions will have to confront the problem of emergent misbehavior. This is not an insurmountable problem, but it is an inevitable one.

For example, IBM has articulated a vision of *autonomic computing*, in which systems self-configure, self-optimize, and self-heal [22]. HP has articulated an *Adaptive Enterprise* vision, in which the IT environment supports rapid changes in business-level strategies and tactics [19]. In many ways, these two initiatives (and those from other companies) overlap, but they differ somewhat in emphasis.

One potential concern about self-optimizing and self-healing systems is that they add additional automated control loops to existing systems with complex behavior. These extra control loops might themselves lead to emergent misbehavior, especially during self-healing actions, which might not be as easily tested as those used in normal situations. (Conversely, Armando Fox points out [15] that the use of control loops inherently exposes measurements of important aspects of system state, which could be used both to detect controller saturation and as partial input to a detector for system-wide misbehavior.)

6.1 Service-Oriented Architectures

Many companies (including HP, IBM, Microsoft, and others) are eagerly adopting the concept of Service-Oriented Architectures (SOAs), in which a set of potentially interchangeable component services (self-contained software agents that interact via network communication) can be composed rapidly to address novel IT requirements. The vision assumes that implementation details of the individual services are irrelevant to the user, and thus SOAs reduce the explicit complexity of a composed application. However, as Gribble points out, "low-level interaction between independently built components can have profound implications on the overall behavior of the system." As a result, an SOA application might still exhibit unexpected complex behavior.

The SOA vision of the future seems to be based on three concepts:

- **Construction by composition**: Complex systems can be constructed by composing well-defined, well-documented, and well-tested components (services).
- **Correctness by construction**: Each composition step is simple enough that it is easy to be sure that the step meets its specification, either by informal inspection or by formal verification.
- **Loose coupling via networks**: component services can be in administratively and geographically distinct places.

These concepts have obvious benefits, which is why SOAs are attractive. However, the "correctness by construction" property might be valid only locally, rather than globally throughout a complex system, once the system has been composed out of independent pieces. The "composition assumption" – that one can build a system with a desired behavior knowing only the behaviors of the components – ignores the possibility of emergent behavior.

In the Millennium Footbridge case, for example, the bridge itself was a carefully designed "component" (and the implementation did, in fact, meet the design specification). The people who walk on it were also thought to be reasonably well-understood components. The interaction between the bridge design and little-known aspects of human behavior was not expected, however. (The tendency of people to synchronize their footsteps with small lateral motions had been reported before, but without any useful quantification [11].)

SOAs will probably introduce distribution into many applications that are currently relatively integrated. As discussed in Section 5.2, the use of networks, especially when they span significant distances, may increase the likelihood of emergent misbehavior, by adding latency to the inter-service interactions.

6.2 Declarative approaches

Coleman and Thompson [10] describe the use of Model-Based Automation (MBA) for the management and construction of IT services. Also, see [19, page 9] for a description of the use of MBA for for application construction. In contrast to the use of imperative scripts for managing systems, MBA uses declarative models for components and their composition. The expected advantage of a declarative approach, as opposed to the traditional procedural approach, is that designers in theory need specify only what they want done, not how to do it.

The paradox of the declarative approach is that, while it should be a more direct way to express the desired goals, it can be quite hard to predict the result of a large number of rules. This can lead to the declarative analog of “spaghetti code,” where the declarative programmer has layered rule upon rule in an attempt to elicit the desired behavior, whereas a procedural programmer would more directly tell the system “do it this way.”

Thus the declarative approach runs the risk of allowing the construction of complex model-driven systems whose behavior is both unpredictable and opaque. Anyone who has tried to debug a set of *sendmail* [1] rules should understand this problem. This is not to say that declarative programming or MBA is a bad idea, but we will have to anticipate and react to the potential for emergent misbehavior in such systems.

One might speculate that there is a critical level of behavioral complexity below which it is feasible to program declaratively, but above which the attempt to do so becomes, in effect, an increasingly chaotic process of “programming by emergent behavior;” that is, an attempt to reach the desired results by manipulating declarative rules, without a predictable connection between rules and results. In other words, there might be limits to system design techniques that attempt to hide the complexity of the underlying problem.

7 Summary

We will never be able to solve all emergent misbehavior problems, especially as system complexity increases. However, we can and should be able to recognize recurring patterns of misbehavior, and to learn enough from past experience to be able to avoid or repair many of the common patterns. Computer systems research has an important role to play, especially in the detection and diagnosis of emergent misbehavior, because of the need for and difficulty of constructing a global view.

Acknowledgments

I would like to thank Mary Baker, Ira Cohen, Armando Fox, Steven Gribble, Mehul Shah, and John Wilkes for their help in preparing this paper, and the anonymous reviewers for HotOS-X for their comments.

References

[1] Eric Allman. SENDMAIL – An Internetwork Mail Router. UNIX Programmer's Manual, 4.2BSD, 2C, Comput. Sci. Division, EECS, Univ. of California, Berkeley,, 1985.

[2] Dave Anderson, Jim Dykes, and Erik Riedel. More Than an Interface–SCSI vs. ATA. In *Proc. FAST*, San Francisco, CA, Mar. 2003.

[3] Mary Baker. Personal communication, 2005.

[4] Aaron B. Brown and Joseph L. Hellerstein. Reducing the Cost of IT Operations–Is Automation Always the Answer? In *Proc. HotOS-X*, Santa Fe, NM, June 2005.

[5] George Candea. Predictable Software – A Shortcut to Dependable Computing? Technical report, Stanford University, March 11 2004. http://arxiv.org/abs/cs.OS/0403013.

[6] Mike Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic systems. In *Proc. 2002 Intl. Conf. on Dependable Systems and Networks*, pages 595–604, Washington, DC, June 2002.

[7] Yvonne Coady, Russ Cox, John DeTreville, Peter Druschel, Joseph Hellerstein, Andrew Hume, Kimberly Keeton, Thu Nguyen, Christopher Small, Lex Stein, and Andrew Warfield. Falling Off the Cliff: When Systems Go Nonlinear. In *Proc. HotOS-X*, Santa Fe, NM, June 2005.

[8] Ira Cohen, Jeff Chase, Moises Goldszmidt, Terence Kelly, and Julie Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Proc. OSDI*, pages 231–244, San Francisco, CA, December 2004.

[9] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, indexing, clustering, and retrieving system history. In *Proc. 20th SOSP*, pages 105–118, Brighton, UK, Oct. 2005.

[10] D. Coleman and C. Thompson. Model Based Automation and Management for the Adaptive Enterprise. In *Proc. 12th Annual Workshop of HP OpenView University Association*, pages 171–184, Porto, Portugal, July 2005.

[11] P. Dallard, A. J. Fitzpatrick, A. Flint, S. Le Bourva, A. Low, R. M. Ridsdill Smith, and M. Wilford. The London Millennium Footbridge. *Structural Engineer*, 79(22):17–35, November 20 2001.

[12] George B. Dyson. *Darwin Among the Machines: The Evolution of Global Intelligence*. Perseus Books Group, 1998.

[13] Kutluhan Erol, Renato Levy, and James Wentworth. Application of agent technology to traffic simulation. In *Proc. of Complex Systems, Intelligent Systems and Interfaces*, Nimes, France, May 1998. http://www.tfhrc.gov/advanc/agent.htm.

[14] Sally Floyd and Van Jacobson. The synchronization of periodic routing messages. In *Proc. SIGCOMM '93*, pages 33–44, 1993.

[15] Armando Fox. Personal communication, 2005.

[16] Martin Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game “life”. *Scientific American*, 223(4):120–123, October 1970.

[17] Steven D. Gribble. Robustness in complex systems. In *Proc. HotOS-VIII*, pages 21–26, Elmau, Germany, May 2001.

[18] Steven D. Gribble. Personal communication, Aug 2005.

[19] Hewlett-Packard. Adaptive Enterprise: Business and IT synchronized to capitalize on change. http://h71028.www7.hp.com/enterprise/cache/7504-0-0-0-121.html, June 2005.

[20] John H. Holland. *Emergence: From Chaos to Order*. Perseus Books, 1998.

[21] Bernardo A. Huberman and Tad Hogg. *The Ecology of Computation (B. Huberman, ed.)*, volume 2 of *Studies in Computer Science and Artificial Intelligence*, chapter The behavior of Computational Ecologies, pages 77–115. North-Holland, Amsterdam, 1988.

[22] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan. 2003.

[23] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX*, pages 1–15, Anaheim, CA, April 2005.

[24] Bhaskar Krishnamachari, Stephen B. Wicker, and Ramon Beja. Phase Transition Phenom-

ena in Wireless Ad-Hoc Networks. In *Proc. Symposium on Ad-Hoc Wireless Networks, IEEE Globecom*, pages 2921–2925, San Antonio, TX, Nov 2001.

[25] MacNeil/Lehrer Productions. The Loss of the Shuttle Columbia: An Online NewsHour Special Report. http://www.pbs.org/newshour/bb/science/columbia/, 2003.

[26] Petros Maniatis, David S. H. Rosenthal, Mema Roussopoulos, Mary Baker, TJ Giuli, and Yanto Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proc. SOSP*, pages 44–59, 2003.

[27] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *CACM*, 33(12):32–44, 1990.

[28] Jeffrey C. Mogul and Greg Minshall. Rethinking the TCP Nagle Algorithm. *Computer Communication Review*, 31(6):6–20, Jan. 2001.

[29] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. on Computer Systems*, 15(3):217–252, Aug. 1997.

[30] Ed Nisley. Emergent Misbehavior. *Dr. Dobb's Journal*, Oct 2004.

[31] H. Van Dyke Parunak and Raymond S. VanderBok. Managing emergent behavior in distributed control systems. In *Proc. ISA-Tech '97*, 1997. http://www.erim.org/ vparunak/isa97.pdf.

[32] Sharon E. Perl and William E. Weihl. Performance assertion checking. In *Proc. SOSP*, pages 134–145, 1993.

[33] K. K. Ramakrishnan and Henry Yang. The Ethernet Capture Effect: Analysis and Solution. In *Proc. IEEE 19th Local Computer Networks Conf.*, Minneapolis, MN, Oct. 1994.

[34] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, Charles Killian, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. in progress, 2005.

[35] Theodore H. Romer, Dennis Lee, Brian N. Bershad, and J. Bradley Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proc. OSDI*, pages 255–266, Monterey, CA, Nov. 1994.

[36] University of Washington Libraries. History of the Tacoma Narrows Bridge. http://www.lib.washington.edu/specialcoll/tnb/, 2004.

[37] John Wilkes. Personal communication, Aug 2005.