Reliability of Complex Services

Sergey Alexandrov

Simplicity is prerequisite for reliability. — Edsger W. Dijkstra
The entropy of a closed system tends to increase over time. – Second Law of Thermodynamics

1. Introduction

In the relatively young history of computers, their capability has grown at an exponential rate – both in hardware (see Moore's Law), and in software. Naturally, along with the increased potential came the greater magnitude of complexity. Where mainframes used to run isolated instructions from single programs, today's conditions require multiple interactions at every level – each CPU multitasks dozens of processes, servers manage their load between multiple CPUs, application tiers take advantage of server farms to manage load, service providers require multiple tiers for efficient services. An industrial-strength service architecture pushes the limits of descriptive reliability models, as the number of nodes and connections in dependency graphs of the service components grows rapidly. Without a doubt, the inherent complexity of service providers, whether it is a brokerage firm, an online retailer, a search engine or a portal, is significant. But on the other hand, the availability of the Internet and global access require these providers to be available 24 hours a day, seven days a week. Reliability being a major concern, much of the marketing literature for service components is filled with terms like "five nines", signifying 99.999% availability – but in real life, we typically find that reliability is much less – anyone with a computer has grown accustomed to encountering various faults, in a manner not applied to older technologies, like land-line phone service or automobiles.

With the service complexity constantly growing, should we admit that it implies decreased reliability? Or can this effect be mitigated? In this paper we examine existing observations on complexity and reliability, and build a model of the relationship between the two. Based on the derived relationship, we argue that complexity does lead to greater unreliability, and operators with limited resources must accept that achieving very high reliability will be intractable.

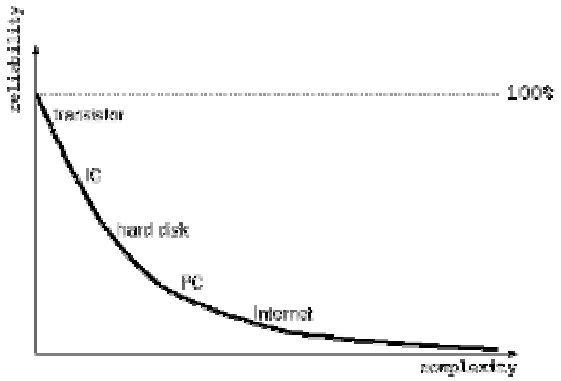

Fig.1 – An illustration of decreasing reliability. [cs.berkeley.edu]

2. Motivation

At a first glance, it seems intuitively obvious that reliability would be inversely proportional to complexity. The more complicated a system is, the more ways there are for it to fail. We see examples of this relationship in our everyday life – a digital watch or a simple calculator survive for decades without malfunctioning, while our desktop machines suffer one malady after next, from inexplicable application crashes to Byzantine faults, etc. We are used to seeing a web page not load, an application not successfully installing, different programs fail to work with each other. Our expectations are lowered, as if software and the Internet were a cable company customer service center, a source for perpetual disappointment.

While the mythical "five nines" reliability is frequently thrown around in hypothetical circumstances (marketing), it is almost never seen in practice. Even large, established service providers often settle for the more reasonable three nines (99.9% uptime). The effort required increases exponentially with the reliability already achieved, so that it frequently becomes financially impractical to push the boundary.

But is complexity the driving force that prevents us from achieving those higher levels of reliability? A typical desktop already presents such a complex, frequently unstable system that even two nines are rare

– so how can a service provider be expected to supercede the client by such a magnitude?

On the other hand, maybe it is possible to achieve more complex behavior and still maintain reliability. If complexity cannot be reduced, then perhaps another factor in determining reliability can be compensated. We don't have to look further than our own bodies for an example of a system that is incredibly complex yet highly reliable – after all, our mean time to failure is significantly longer than that of any computer system.

3. Overcoming Complexity?

Arguments can be made that complex systems are not necessarily unreliable. If it is possible to achieve high reliability for some complex systems, then it should be possible to achieve high reliability for an arbitrary complex system.

Let's take for example processor design – micro-miniaturization in circuit design has led to the development of increasingly complex CPUs, now utilizing billions of transistors per circuit, whereas 30 years ago the number barely hit four digits. So while their architecture has grown more complex (higher precision, parallelism, pipelining, etc), modern processors are not significantly less reliable – typically the CPU is the least likely component to fail. This is obviously directly counter to an assumption of an inversely proportional relationship between complexity and reliability – especially if one defines complexity as proportional to size.

On the software end, the National Software Testing Labs found that Windows operating systems have achieved a greater MTBF (from 216 hours for Windows 98 to 2893 hours for Windows 2000), despite an increase in system complexity (measured, for example, in the number of available system calls) [15].

In another example, all modern cars now rely on central computers to control both primary and auxiliary functions. Yet according to a UK study [8], software faults constituted only 0.1% of all recalls, and a contributed to a worst-case rate of 0.2×10^{-6} injuries per hour of driving – in other words, a MTBF of 570 years. Similarly, today's large passenger airplanes rely on sophisticated auto-pilot software for all phases of the flight, including take-offs and landing, as well as other functions. Although the procedures require careful analysis with a high cost of failure, a study [7] assessed the MTBF for software faults at 1826 years.

Reliability research and development has resulted in a bevy of design, testing and fault tolerance methodologies designed to increase uptime. More programming languages feature managed code, modular design has led to heavy utilization of reusable, well-tested stable components (as is often the case, we can draw useful parallels from biology – a study [11] suggests that in nature, and in technology, more complex systems attain reliability by simplifying their sub-systems – by limiting components to small, highly specific tasks), and development shops often feature extensive automated testing and code coverage tools. With cheap hardware we now have affordable options like redundancy and failover.

Indeed, there are some examples of very reliable complex services – for example, Google.com, which provides efficient web searching under extremely high load, or NASA's web site, which claims 99.995% uptime. However, other large well-know service, like eBay or Amazon, are known for occasional failures, despite large investments to prevent them.

4. Complexity

In order to evaluate the effect of complexity, we must first define it. What is complexity, and how do we measure it? Generally complexity is a function of the magnitude of the state space represented by the system in question. The larger it is, the more possibilities, and the more possibilities, the more chances for unforeseen or untested circumstances. It is also important to note that complexity of a service is not limited to just code complexity, but is a function of several factors which we discuss here.

4.1 Code Complexity

The most studied, and perhaps the most easily quantifiable, type of complexity is code complexity (we'll call it C^{CODE}) – a measure of how complicated is the actual software. The earliest and weakest measurement was the number of (non-comment) lines of code. This however suffers from numerous flaws, such as the effect of formatting, language-specific syntax, programming style, etc. The Halstead Software Science metrics [1] take a more sophisticated approach, by counting the number of unique operands and operators, and the frequency of their usage. By examining intrinsic properties of the program, particularly enumerating instances where a potential error may occur, these metrics come closer at providing a better complexity measurement, but thresholds are still arbitrary, and subject to specific language implementation.

A more sophisticated, widely used metric for software complexity is the McCabe cyclomatic complexity, along

with its derivatives [2]. Rather than looking at operands, this metric addresses control flow, building a graph of possible execution paths. A higher number of distinct independent pathways through the graph implies a higher potential for error – in other words, the programmer and the tester must cover more cases, and the maintainer must consider more situations when making changes.

An even more fine-grained approach is the cognitive functional size metric (CFS), which evaluates a module based on the cognitive weight (a difficulty rating, so to speak) of each of its basic control structures, modified by its inputs and outputs [6].

4.2 Structural Complexity

The above metrics however only apply to single code modules, since interaction between modules is not taken into account. Today's systems are highly modularized, involving many interconnected, distinct components, and a complexity metric ignoring this will by naturally incomplete. Structure complexity (C^{STRUC}) metrics can be applied for thus purpose, like IFIO (informational fan-in/fan-out) [3], which calculates the degree of component interaction (fan-in is the number of components calling a particular component, fan-out is the number of components called by that component). Or, a metric like Card's System Complexity, SYSC [4], can be used, which totals up structural and data complexity (the latter evaluating the amount of data passed between components) over the whole system. More interestingly is RSYSC, an average of that metric, which allows systems of different sizes to be compared – one of the few metrics to give a system-independent scale. A more recent metric specifically derived for component-based system is the Interface Complexity Metric, ICM [13], which takes into account the defined API complexity (particularly the signature and corresponding constraints).

4.3 Configuration Complexity

Configuration complexity (C^{CONF}) is a measure of how extensive the setting space is for the given system. For a typical online service, this means web server settings, app server settings, database settings, router settings, firewall settings, etc. The longer the configuration files, the more options for each setting, the larger the setting space, and the greater the overall configuration complexity – and therefore the potential for operator error. Unfortunately, research for configuration complexity is lacking in proven metrics, with only a couple of proposed benchmark models [12].

4.4 Environment Complexity

Finally, we must address the complexity of the environment in which a system must run (C^{ENV}). While a highly specialized, closed system – let's say an electronic fast-food ordering kiosk – does not suffer from environment complexity, a typical online service has to take it into account. This includes the diverse range of potential clients (web browsers, plugins, operating systems, etc.), coexistence with other systems (virus scanners, other unrelated web applications, etc). Client-side applications in particularly suffer from high environment complexity, due to the high number of possible combinations of coexisting software.

5. Reliability

Just as we have done with complexity, we must define how we measure reliability. Once again, many different metrics exist, but a small set represents those used most frequently in the industry. Reliability can be measured in terms of fault rates – either a frequency of fault, or more descriptively, a distribution of faults over time. Alternately, one can measure the mean time between failures, or MTBF, which simply calculates the average time between system faults. Related to this is the mean time to recovery, or MTTR, which measure the average time between a fault occurrence, and the restoration of the system to normal operation. Both of these metrics are reflected in a primary metric when it comes to service operators – uptime, or the percentage of time the service is available for normal operation. Financially, this means the percentage of the time revenue can be incurred – allowing the operator to calculate the amount of revenue lost due to faults.

6. Putting It Together

It is generally accepted that software complexity leads to an increase in the number of faults. One of the first formal theoretical ties between complexity and reliability is Lusser's law, which states:

> *Reliability of a series system is equal to the product of the reliability of its component subsystems, if their failure modes are known to be statistically independent.*

In other words, greater size equals greater complexity equals lower reliability. We can apply it to software by breaking down code into basic components, so a given execution path (or, the data path) must follow them in series. If we accept the independence assumption, then the expected failure rate is directly proportional to the length of the execution path. Furthermore, while shared data means software components are not independent in terms of rate of failure, an increase in the failure rate of

one component does not decrease it in another, implying an even lower reliability in the non-independent case. Thus our first assumption is that the greater the system, the lower the reliability.

So how can we more precisely relate complexity and reliability? First, let's define a new complexity measure, C^{TASK} – the complexity of the required functionality of a system. The object is to describe the minimal complexity of a given system specification. This provides the optimal baseline for the actual implementation:

$$C^{CODE} = k_{impl} C^{TASK}$$
$$k_{impl} \geq 1$$

Here, k_{impl} is some factor signifying the efficiency of the implementation. Its value is affected by the quality of the code design – the simpler the implementation, the lower the value. Since by definition C^{TASK} is the optimal value, it has a lower bound of 1).

There have been many empirical studies showing a correlation between code complexity and error rates, a few of which are shown in Table 1:

Author	Conclusion
Gremillion (1984)	Lines of code was determined to be the most accurate predictor of repair request volume. Repair request increased as a function of size and frequency of use-related to complexity and age.
Lind and Vairavan (1989)	Lines of code correlated with the development effort as well as their more sophisticated standard complexity metrics.
Henry and Kafura (1981)	Complexity measure highly correlated with number of changes made, suggesting that complexity metrics may predict error rates.
Card and Agresti (1988)	Changes in measured complexity accounted for 60% of the variation in error rate.
Basili and Perricone (1984)	Larger modules appear significantly less error prone (per LOC).
Shen et al. (1985)	Smaller modules have a higher rate of errors per LOC than larger modules. Metrics related to amount of data and the structural complexity (number of loops, conditional statements, and Boolean operators) proved to be the most useful in identifying error prone modules at the earliest stages of testing.

Table 1 [9]

The overall trend shown here is that more complex code generates lower reliability. Furthermore, the last two studies noted indicate that smaller modules suffer from higher error densities – an unfortunate trend, given that modern design patterns dictate a significant use of encapsulation, modularization, tiered architectures, and other approaches that require that code is broken into small components, in comparison to the monolithic programs of the past.

Another study [20] showed that there was a significant correlation between complexity (here measured as cyclomatic density) and the amount of subsequent maintenance effort (complex modules required more changes/repairs), even though there was no correlation between size and maintenance effort – confirming the proposition that complexity is not driven by size when predicting reliability.

While effects of complexity can be modified somewhat as discussed in Section 6, the correlation still persists. In [10], a study was performed on IBM and Nortel systems, evaluating multiple measures of complexity against defect rates. It was found that while the most complex modules were not necessarily the most defect-prone, and vice versa, they tended to be ranked among the higher-defect clusters.

Finally, it was instructive to look at recommendations of a study by the Canadian Space Agency, whose purpose it was to evaluate software metrics in terms of their effectiveness in predicting reliability. Table 2 shows the recommended complexity metrics.

Metric	Reason for Recommendation
LOC (lines of code)	Multiple studies validated correlation between LOC and number of faults.
Cyclomatic Complexity	Multiple studies validated correlation between the CC number and number of faults.
Card's System Complexity	Large industrial experiment validated correlation between complexity and number of faults.

Table 2 [16]

Taking these results into account, we can state that both code complexity and structural complexity affects predicted fault density (FD^{PRED}):

$$FD^{PRED} \propto C^{CODE}$$
$$FD^{PRED} \propto C^{STRUC}$$

The actual fault density depends on the amount of testing and debugging – the more bugs are identified prior to release, the less errors will occur during operation.

$$FD^{ACTUAL} \propto \frac{FD^{PRED}}{QA}$$

Here, *QA* is a testing metric, measuring coverage – how much of the code has been tested, and to what extent. A typical coverage metric is, for example, the number of test cases evaluated / number of requirements. The more complex the task, in other words, the more test cases are needed:

$$QA \propto \frac{k_{test}}{C^{TASK}}$$

Actual reliability however is not necessarily measured in fault density – as mentioned earlier, it is really the system uptime – the portion of time that the system successfully services the user – that is of primary concern. Therefore, modern services employ a high degree of fault tolerance, including redundancy, replication, recovery, retry, and other mechanisms designed – ensuring that many faults are hidden from the user. So uptime is inversely proportional to the fault density for those faults that the system cannot hide from the user:

$$Uptime \propto \frac{k_{ft}}{FD^{ACTUAL}}$$

This is confirmed by a critique of software reliability models [17], which concluded that it is both code complexity and testing effort that contribute to actual fault densities.

So far we have only taken code complexity into account. However, back in Section 3 we defined other complexity factors that can contribute to faults. Configuration complexity increases operator errors (the more configuration options, the more opportunity for misconfiguration), environmental complexity increases potential for errors due to environmental factors (for example, the service-to-client network link failing), etc. So the more complete relationship would have to include these factors into consideration:

$$Uptime \propto \frac{k_{ft}}{FD^{ACTUAL} + FD^{OPERATOR} + FD^{EXTERNAL}}$$

Studies have shown that software causes only a part of the total errors in systems from Internet services [18] to missile systems and PSTNs [19] – a significant amount was due to operator errors, which tended to have a large effect on the resulting reliability. So we can express this as:

$$FD^{OPERATOR} \propto \frac{C^{CONF}}{k_{conf}}$$

Here, k_{conf} represents the resources spent on operator training and support structures (knowledge bases, etc).

Finally, a similar breakdown of environmental errors:

$$FD^{ENV} \propto \frac{C^{ENV}}{k_{env}}$$

Again, here k_{env} represents the resources spent to stabilize the environment (like hiring network support personnel).

Putting the whole thing together:

$$Uptime \propto \frac{k_{ft} k_{test}}{C^{TASK} k_{impl} C^{TASK} + \frac{C^{CONF}}{k_{conf}} + \frac{C^{ENV}}{k_{env}}}$$

While this is of course a generalization, this does signify that task complexity plays a significant role in reducing uptime. Given a system requirement, with the corresponding minimal task complexity, the above relationship gives us the following paths to increasing uptime:

Approach	**Caveats**
Increase k_{ft} (fault tolerance)	Adding fault tolerance mechanisms tends to increase non-task complexity (more components to configure, for example), and since those mechanisms are also subject to failure, it introduces a feedback effect into the complexity-to-reliability relationship. Additionally, not all faults can be successfully hidden.
Increase k_{test}, (test coverage)	Exhaustive testing for sufficiently complex systems is intractable given finite resources. Furthermore, studies have shown that additional testing has a diminishing return on the number of bugs detected – at some point, further testing becomes financially and logistically imprudent [14].
Decrease k_{impl} (simplify/optimize implementation)	Has a hard lower bound (see definition).
Increase k_{conf} (provide better operator training, supply better documentation).	Humans are by definition fallable, so the benefit is limited.
Increase k_{env} (add resources to stabilize the	The benefit can be limited – for example, for a web-based service, not

environment).	all instabilities can be overcome (like a network link going down outside of the operator's control).
Decrease C^{ENV} (reduce environmental complexity)	Not always possible – web-based services for example must accept the imperfections of the Internet as their requisite environment.
Decrease C^{CONF} (reduce configuration complexity)	Limited because most systems require some minimal configuration state space for the system to operate.

It's also important to note that each approach involves the expenditure of additional resources – money, manpower, time. Since such resources are finite, much of reliability research is focused on determining the optimal application of these steps.

7. Conclusions

From the relationship derived in Section 6, we must infer the following conclusion:

Given fixed resources, more complex requirements will result in less reliable systems.

Unfortunately, the trend over time is that requirements become more complex. Clients want more features, more options, more functionality, all of which results in greater task complexity. So just to maintain reliability given such an increase, greater resources (development time, testing time, better fault tolerance infrastructure, etc) must be expended.

Let's return to a previous example of reliable systems to view them from the perspective of the above relationship. Both automobile and flight control software (and similarly space control systems) benefit from extensive research, design optimization, testing, and fault tolerance mechanisms – so k_{impl}, k_{test}, and k_{ft} are all optimized – a the expense of significant time and resources. Furthermore, they are closed, specialized systems, designed for minimum operator input, therefore C^{ENV}, C^{STRUC}, and C^{CONF} are all kept low. All of this adds up to high reliability.

A typical online service, however, has limited resources, and suffers from increasing environmental complexity. A typical service faces a myriad of potential complications beyond its inherent software complexity. It must serve a wide variety of clients, running on different operating systems and different conditions. It must rely on networks that are outside of its area of control, and it must deal with technology partners (for example, stock quote providers, trading services, financial information clearinghouses, etc) that are also outside of its scope and are subject to their own limited reliability.

Therefore for service providers with limited resources, the growing complexity will prevent them for achieving very high reliability for their offerings.

The only potential hope is that the growth of required task complexity will halt, or at least will abate enough to where the rate of improvement (and availability) for best-of-breed approaches to implementation, testing, and fault tolerance can overtake it.

8. References

1. Maurice H. Halstead, *Elements of Software Science*, Elsevier North-Holland, New York, 1977.
2. Thomas J. McCabe, A Complexity Measure, *IEEE Transactions on Software Engineering*, SE-2 No. 4, pp. 308-320, December 1976.
3. S. Henry and D. Kafura, Software structure metrics based on information flow, *IEEE Transactions on Software Engineering* 7 (5), 510—518, 1981.
4. D. N. Card and W. W. Agresti, Measuring Software Design Complexity, *The Journal of Systems And Software* 8, 3, 185-197, June 1988.
5. J. C. Munson, Software Faults, Software Failures, and Software Reliability Modeling, *Information and Software Technology*, December, 1996.
6. Yingxu Wang and Jingqiu Shao, Measurement of the Cognitive Functional Complexity of Software, *icci*, p. 67, Second IEEE International Conference on Cognitive Informatics (ICCI'03), 2003.
7. M.L. Shooman, Avionics software problem occurrence rates, *issre*, p. 55, The Seventh International Symposium on Software Reliability Engineering (ISSRE '96), 1996.
8. Michael Ellims, *On Wheels, Nuts and Software*, SCS 2004: 67-76, 2004.
9. R. Banker, S. Datar, and D. Zweig, Software Errors and Software Maintenance Management, *Information Technology and Management*, v. 3, nos. 1/2, pp. 25-41, January 2002.
10. Akif Günes Koru, Jeff Tian, An empirical comparison and characterization of high defect and high complexity modules, *Journal of Systems and Software* 67(3): 153-163, 2003.
11. O.A. Bogatyreva and N.R. Bogatyrev, Complexity in living and non-living systems, *Proc. Of international TRIZ conference*, p. 16/1- 16/6, March 2003.
12. Aaron B. Brown and Joseph L. Hellerstein, An Approach to Benchmarking Configuration Complexity. SIGOPSEW 2004 - 11th ACM SIGOPS European Workshop. ACM SIGOPS, July 2004.
13. N.S. Gill, and P.S. Grover, Few important considerations for deriving interface complexity

metric for component-based systems. *SIGSOFT Softw. Eng. Notes* 29, 2 (Mar. 2004), 4-4.
14. B. Littlewood and L.Strigini, Validation of Ultra-High Dependability for Software-based Systems, *Communications of the ACM* 36(11): 69-80, 1993.
15. National Software Test Labs, *Comparison of the Reliability of Desktop Operating Systems*, www.nstl.com, February 2000.
16. M. Frappier, S. Matwin, and A. Mili, Software Metrics for Predicting Reliability, *Software Metrics Study: Technical Memorandum 2*, 1994.
17. E. Norman Fenton, A Critique Of Software Defect Prediction Models, *IEEE Transactions of Software Engineerting*, Vol. 25, NO. 3, May/June 1999.
18. D. Oppenheimer, A. Ganapathi, and D. A. Patterson, Why do Internet service fail, and what can be done about it?, *Proc. USENIX Symposium on Internet Technologies and Systems*, 2003.
19. A. Brown and D. A. Patterson, To Err Is Human, *Proceedings of the First Workshop on Evaluating and Architecting System Dpendability* (EASY '01), 2001.
20. Chris F. Kemerer and S. Slaughter, Determinants of Software Maintenance Profiles: An Empirical Investigation, *Journal of Software Maintenance*, v. 9, pp. 235-251, 1997.