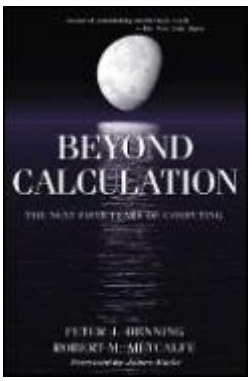
Beyond Limits

Natural Limits?

Contents

ACM Permission



Note: This paper has been published in "Beyond Calculation: The Next Fifty Years of Computing"; ISBN 0-387-94932-1.

Introduction

The first million was easy. Computers have improved by a factor of millions in price, performance, capacity and capability in their first fifty years.

We've come to expect this improvement. Memory prices, for example, halve every 18 months (according to Moore's Law), CPU's get faster and software does more. Today's desktop computers are far more powerful than the mainframe computers from 25 years ago.

Are we running into natural limits?

In the excitement about what we've accomplished we should remember that we have not fulfilled many of the promises of very intelligent machines. If anything, we've come to see the computer as a fancy calculator or word processor and little more. Shouldn't we expect more from these systems? More to the point, why are computers so *hard to use?*

While we might reach limits on particular technologies, we are far from the limits on what we can do with computers. The pace of change is limited only by our ability to innovate. This pace has been accelerating because the computer itself is our key tool. As we improve the computers, we increase our ability to improve them.

As we innovate, we keep changing the nature of computers. The "computer" itself is a device that performs computations. The next stage shifts the focus from what we can do in the computers to what we can accomplish with them as elements in the larger infrastructure. The computers themselves will "disappear into the woodwork". Our challenge is to learn how to master this new arena – one in which we are not writing programs but adding intelligence to everything around us. The limit is in our ability to manage complexity. It is a world in which resiliency is more important than perfection. A resilient system is one that can continue to function in the midst of the chaos and failure which is the norm.

As a developer, I'm very concerned with how we evolve computing. The challenges of complexity are overwhelming. When we back and observe the history of computing,, the individual changes fade into the grand forest of innova-

tion. Stepping further backing, computing itself is a part of as opposed to apart from the evolutionary process of innovation.

Step too far back and we fail to see that the evolution of computing has not been uniform. What we got isn't necessarily what we asked for. But it is in the nature of systems to exploit unexpected advantages.

The history of computers has also been rife with failed promises and frustrated expectations. Yet these tend to be forgotten in the excitement of what we can – almost – do.

Once again, we are at a transition point. (When are we not?). We are leaving the confines of the isolated computer and becoming the infrastructure; an infrastructure of intelligent elements. And we have no idea where it will lead but we can be sure that the pace of change will continue to accelerate.

The First Fifty Years

Rapid Change and Innovation

When four-function calculators were first introduced they cost $1000 (in 1997 dollars) and now they are given away free, the cost being covered by the advertisements on the back. Not only does a wristwatch contain a computer, but it plays music and, soon, may also be a telephone.

The first half century of computers has been a period of rapid advancement in hardware and software design.

This rapid pace was driven by many innovations. Core memory was created to meet the needs of the early computers. Semiconductors were invented separately but were crucial to the ability to create large systems. But even before transistors became widely available, tube technology had been advanced beyond anything believed possible in the days of radio.

Equally important were the improvements in software. Operating systems were created to make efficient use of expensive hardware; tools were created to make programming simpler. These innovations used existing hardware existing. Only after the software was available was the hardware tuned to meet the needs of the software. The improvements in software are characterized by Corbató's law which states that the number of lines of code is the same, independent of the language used. The more we can leverage programmers with tools that make it easier to express their algorithms, the more productive they are

Projects that required innovating in too many areas at once were unlikely to succeed. The IBM 360 and the Multics project were notable exceptions and both experienced long delays in delivering on their promises. The lessons of why these projects were so difficult are still relevant today. Fred Brooks' The Mythical Man Month applies to any complex system, not just a large pride of programmers.

After the success of these first fifty years, are we running into the limits on what computers can do? What if we can't make circuits much smaller than they are now, what if we can't dissipate the heat? The size of an atom hasn't shrunk and you need at least one to make a wire.

This mood of uncertainty about our ability to overcome limits is not new. In 1798 Thomas Robert Malthus wrote his Essay on the Principle of Population as It Affects the Future Improvement of Society. His basic claim was that we were doomed to starve because population increases exponentially but food sources increase linearly.

This pessimistic view fails to take into account the nature of change and innovation. It is necessary to be flexible and build upon the innovations available rather than just improving already available technologies. While there have been many technological advances in food production, it has also been necessary to improve distribution and to limit population growth. The innovations build on each other but are not rigidly dependent upon each other. If we have better distribution we can bring food from farther away or we can improve the local crop yield. If we have higher agricultural productivity we can move to the city.

With computers we have an additional element – the computers themselves are direct agents in the process of innovation.

The Marketplace and Creative Solutions

The evolution of personal computers was accelerated because electronic spreadsheets appealed to investors. The marketplace directly funded the development of the technology. This is a particularly dramatic example of the value of a marketplace in driving innovation.

The size of the marketplace was also important for a variety of approaches to coexist and flourish enriching the store of concepts available for reuse.

The development of graphics processors is a good example of a sufficiently large specialized market. 3D graphics processors can be targeted at consumer games and then used for commercial visualization while retaining the low price of the large consumer marketplace..

Innovations are typically in service of a need. In the case of communications, a major need is increased bandwidth. We can increase bandwidth by improving signal processing or by compressing the data. As we run into the limits of the signal speed, we improve compression. Voice compression reduced bandwidth requirements from 32Kbps to 9.6Kbps or less in a few years. Some of this has

been due to faster processors and some due to algorithms such as modeling the constraints of the human mouth.

To the user, communications simply became better (Cheaper? More capable? Faster?). What seems to be a uniform process of improvement is composed of disparate elements.

The Evolving Nature of Programming

The dramatic changes in hardware often obscure changes in software. We've gone from wiring plug-boards for each calculation to drawing a description of what we want them to do. In some cases the computer watches what you do and tries to offer its own suggestions which not always appreciated.

Using the computer itself as the tool for its own programming has been central. Assemblers allowed machine instructions to be coded symbolically; later compilers converted program descriptions into machine instructions; and development environments became available to manage the process. Note that these terms: "assemblers", "compilers" and even "computers" come from human jobs of similar functions but the machines have become the agents for these tasks.

The meaning of programming has itself evolved. Initially the focus has been on coding the steps involved in solving an identified problem. As the tools become more powerful, the coding aspects have become automated and the focus has been on the description of the problem itself. In fact, original programming languages were called Automatic Programming tools since they automatically generated the program from the description – we now refer to the description as the program. But as our capabilities have grown so have our expectations and thus our requirements. The term coding moved from specifying machine instructions to writing in a language such as COBOL or FORTRAN. Later generations of tools allowed for describing the desired result rather than an algorithm. One might specify a report by giving a sample rather than the details of how to construct the report.

The challenge has shifted from providing the professional programmer with tools to providing the "users" with the tools to directly interact with the computer. The original users of FORTRAN saw themselves as, and were, scientists and engineers solving their own problems. As their needs grew they had to choose between focusing on programming computers or on their area of professional expertise. For many, programming was more seductive.

As we've expanded the set of "programmers" to include, potentially, anyone using a computer, we've also changed the nature of programming. Rather than specifying a series of steps, one can give examples or a description of what should be done rather than the detailed steps for how to do it.

Even if these users approach the computer as an improved version of an earlier device such as a typewriter, the real power comes from understanding the new capabilities. Rather than "typing", one uses rule-based "styles". Instead of being concerned with the attributes of each "section head", one tags (or "codes") each header as such and then can set the properties of all heads and, perhaps, subheads as well, at once. Later one can add some rules to describe what happens if the head is stuck at the bottom of a page or how to handle odd pages differently from even pages. It is not necessary to have this understanding to use the computer to type, but those who do are rewarded with a more effective tool. Of course, the software vendors are trying to garner the largest possible market and so have an interest in making the capabilities more accessible. Features understood by only a few are liable to be discarded.

Likewise, the spreadsheet is not just a digital analog of a calculator but a tool that allows for experimentation. (Only later did it become a presentation tool, but that's another story). In fact, the electronic spreadsheet derived its power from allowing a user to specify an algorithm by "doing it" and then being able to repeat the operation with new values. The name "VisiCalc" emphasized the visibility of this process. We were, in fact, getting people to program without realizing that's what they were doing.

It is this ability to use the computer as an agent by "programming" it with behavior that is central to the power of computing. It is important to realize that we have converted the user into a programmer just as the phone dial converted people (users?) into phone operators. In the 1930's there were some estimates that by the 1950's we'd need to have everyone to be a phone operator in order to satisfy the demand. The effect of automating the phone system can be viewed as not eliminating phone operators but making everyone a phone operator.

Requiring a separate class of programmers who translate user requirements into algorithms is not only expensive but ultimately frustrates our ability to make effective use of the technology. It is this requirement for the specification of behaviors and effective algorithms that is at the heart of the societal change. Just as there weren't going to be enough phone operators, there aren't enough programmers to add all the little bits of intelligent behavior we are going to expect of the infrastructure. And it is this limitations imposed by this need to specify behavior that is part of the upcoming challenge.

The ability to be descriptive is an important twist on programming in both the spreadsheet and the telephone system. Rather than specifying programming as a series of

step by step operations, the user describes the behavior in a "language" that is shared with the computer. This allows the computer to do more than blindly execute the steps. It can also explain what is happening and recover from many possible problems. But there is still enough freedom left to the user to "reach a wrong [telephone] number" or specify an incorrect formula on the spreadsheet.

From Programming to Problem Solving

One way to characterize problem solving is as the process of making the complex simple.

Computer systems come from a heritage of extremely complex systems built with seeming perfection. Error rates in the trillionths and better are not unusual. Initially this was achieved by careful engineering. Programs were carefully audited to be seemingly bug-free. We even had the notion of proving programs correct.

At best, one can prove that two representations of an algorithm are equivalent but that doesn't address the question of whether the program meets a vaguer requirement. The question is whether the program works properly in service of some larger goal. There may, in fact, be multiple conflicting goals.

Rather than proving programs correct, we must make them simple enough to understand.

The Copernican heliocentric solar system was more than a mathematical reformulation of Ptolemaic system with the Earth at the center. It represented a better understanding the motion of the planets. The heuristic is that the simpler solution is better (Occam's razor). We can take this one step further and argue that simplification is our goal.

But this begs the question since it just shifts the problem to finding the right representation which is unsolvable in the general case. Both because it reminds us that the nature of the solution is a function of the context in which the problem is being solved (ambiguity) and simply because it is simply a restatement of general problem-solving.

But there are elements of a solution here. While we can't necessarily find the right decomposition, we can iterate on the problem and redecompose the problem as we improve our understanding. In practice, if we start out with an initial structure we can recompose the set of elements, or objects, as our understanding is refined. In terms of object-oriented system, as long as we have control over our set of the problem space, we can iterate on the system design. This is an effective technique but it becomes more difficult as the scope increases. Fred Books addressed some of the implications of scale in the Mythical Man Month. The same issues that arise with adding people to a task also arise when building a large system where iterating on the whole design becomes increasingly difficult.

When we have independent interacting systems we don't necessarily have the option of recomposing them. This places a premium on getting an effective representation the first time but, inevitably, the initial solution will need to be adjusted as the situation changes. To the extent we can, we must be prepared for such change.

Communications and computing

The impact of the Web has been dramatic – more than the Internet itself. In the "calculator era" computers stood entirely alone. They took input on paper tape or cards and produced results on a printer or maybe punched out some result for later use. That was a long time ago. In the 1960's time sharing became common and in the 70's and 80's, the Arpanet, later the Internet, started to link systems together. Local Area Networks (LANs) became common in the 80's. The impact of the Web was dramatic because it brought connectivity to the center of computing.

Like VisiCalc, the Web came about at just the "right time". More to the point, there was a waiting ecological niche. The Internet was sufficiently ubiquitous to be the basis for a global infrastructure. What was needed was an effective way to name elements in this network. The key to the URL (Universal Resource Locator) is that it is a pragmatic name that is not only where a resource is but how to access it. The "http" in the URL could also be "ftp", for example, for File Transfer Protocol. Thus we absorb the old protocols into the new without giving up any of the old capabilities. A graphical browser (Mosaic) for the widely available consumer platforms made the power available beyond the scientific community that the Web was originally created for.

Once again we have a positive feedback cycle with the Web growing in scope because of the Web. Not only do we have the tight loop with the Web being the means of improving the Web – each iteration brings in more participants and their contributions. The result is a very rapid growth, or a hypergrowth.

The Internet protocols were themselves built upon simple standards with the main tools being a terminal program (Telnet) and the text editor. The Web came about during a period when the Internet seemed to be getting saturated and was suffering from slowdowns and other results of overextension.

Yet the Internet is now much larger with many times more users. Of course, there are the standard predictions of collapse. The difference is that the Web has transformed the Internet from a tool for the cognoscenti to one of the fundamental engines of society.

It won't fail because we can't let it fail. Our ability to learn to be resilient in the face of failures will allow us to avoid the collapse that is characteristic of rigid systems.

In the earlier example, we saw that there was a tradeoff between bandwidth and communications speed. With the Internet we have another communications tradeoff in the ability to use a very jittery and not fully reliable medium (the Internet) as an alternative to the well-engineered, isochronous PSTN or Public Switched Telephone Network. The impact will be profound because we're selling telephony components linked together with software rather than a single "dialtone" service. The threat to the phone network is not just in the dramatically better economics of the Internet, it is also in the ability to define new telephony services purely in software.

Generations

We have a tendency to group together a set of changes into arbitrary "generations". There is a reality to this in that small changes aggregate to larger trends. Operating systems for mainframes serve to dole out scarce resources. Minis, being less expensive, were tuned for particular purposes. Personal computers started out as an extreme simplification of earlier computers for a very low price with limited utility.

Generational change serves a necessary function of clearing the underbrush of complex ideas so that new ideas can flourish. The radical simplification of computers in PC's has allowed the growth of new operating systems with great emphasis on the ad-hoc integration of applications. The term "application" itself represents a shift from emphasis on the isolated program to its role in service of a task.

Though the various hypergrowth phenomena seem to come just in time out of nowhere, if we look closely we can see their antecedents. VisiCalc had screen editors and calculators, the Web had the WAIS, FTP, Telnet and Gopher, simpler access tools. CISC hardware had the RISC experience to draw upon. The hardware, software and, especially, networking growth are building upon themselves.

For the last twenty five years the Internet has been growing in importance until it was unleashed by the Web. The interactions between applications over the Internet are an extended form of the cooperation among applications within the personal computer itself. This is setting the stage for the next change in the nature of computing.

Becoming the infrastructure

We are in the midst of a fundamental change in the nature of and the role of computing. We are creating a global communications medium that supports digital connectivity among the computing agents throughout the world. We are also deploying bits of intelligence throughout the infrastructure.

The growth of the Internet (often confused with the Web which is just a set of capabilities riding the Internet) is dramatic in its own right. What is less obvious is the growth of intelligent elements such as light bulbs that implement their own lighting policies; or cars that use a local area network to coordinate their components and the global network to report diagnostic information and get traffic updates.

The traditional approaches to system design posit that there is a system being designed. We are adding to a complex system without any overall coordination. Once again we've introduced major sources of complexity without the corresponding means of dealing with it. We need to learn how.

In a sense, the overwhelming scope of the problem contains the seeds of how to approach a solution. Techniques that seemed sensible in a well-understood system just don't work. There is no single version of software to be updated. But, alas, cleverness allows us to keep up the illusion that we are still operating in the old world of self-contained systems. Remote Procedure Calls allow us to pretend that we are invoking a local subroutine when we might be using arbitrary resources on the network.

The deception fails when there is something goes awry, even something as simple as a delay. The result appears as just one more case of computer unreliability rather than as a symptom of a fundamentally flawed the approach embodied in the programming tools.

The file system interface for a disk drive doesn't have the semantics for reporting that the network cable fell out. And the network itself fails to detect this since it is a mechanical problem and not a "network" problem.

The problems of naive extensions of existing solutions should sort itself out as we develop alternative approaches which focus on the interactions between systems New methodologies will have to be resilient enough to survive in a constantly changing, inconsistent environment by bending rather than failing.

Interactions

What happens between or among applications can be more important than what happens within them.

A system consisting of a million well organized parts is not complex in the sense that a system of a hundred autonomous systems is. The real measure of complexity is not the number of elements but the number of (nonuniform) interactions. The way of dealing with this complexity is to

reduce the interactions. In psychology this is called "chunking" and humans seem to be able to process less than ten such chunks at once. This represents an extreme, but effective simplification of the world. In programming this can be a matter of finding a representation that allows us to factor the problem into subproblems with limited interactions.

Techniques such as structured programming, modular programming and object oriented programming (to observe the evolution of the concepts) have been attempts to provide the programmer with structuring mechanisms. But they have mostly focused on interactions within a set of programs. By finding the proper structuring of a program we can decompose it into elements and then manage the interaction among the elements. Solving a problem by finding an effective representation is a recurring theme.

The challenge is not simply to create programs in isolation but to create independent systems that interact. The interactions are not preplanned. Furthermore, failures must be bounded and their propagation must be limited. In the world of the Internet, all systems are potentially interconnected. Ideally local failures do not lead to failures of the entire system. Within a single computer, we can be very ambitious in designing interactions among systems and must sometimes completely reset the entire system to clear out the knots that form among these interactions. This is not an option for the systems that form the global infrastructure.

What makes the problem of managing the interactions even more difficult is that the systems are not necessarily well-managed – if they are managed at all. Increasingly "programs" are being provided by people who do not even view themselves as "programmers" and the linkages are not well understood. Mix in a little Internet and we have a powerful brew.

Scalability

In order to scale systems it is necessary to be able to regenerate reliability. Normally when you multiply probabilities of success, the result is to decrease the reliability at each stage. We've been able to defeat this phenomena by having a way to "understand" the constraints of a system and use this understanding to regenerate the likelihood of success. Active elements operating independently without sufficient defense against failure of other modules and without a description of how they should work together lack this regenerative property.

We've pushed the limits of hardware by determining how to make locally reliable devices. Initially, for example, we could use a modem to send data across the country as 10 characters a second simply by shifting between two frequencies for the 1's and 0's.But now we send 28.8Kbps (or more!) across channels designed for 3Khz voice we are using complex algorithms to make up for the unreliability of the channel. We hide this complexity within the modem.

Problems that are not amenable to a localized attack are much more difficult to solve.

An important change is to shift from algorithmic programming (traditional) to descriptive programming. The description limits the "program" to the common understanding between the describer (or user) and the computer. Describing the interactions between elements allows an observer (the computer) to assist in maintaining the integrity of these interactions and in regenerating reliability.

The description is only in terms of the common understanding and, like the railroad, can only go where track is already laid. We are thus limited by the speed at which we can lay tracks or define the language. It is the nature of the frontier for the attention to be focused at the leading edge. In a sense, the trailing standards setting is a form of track laying. As with railroad tracks the descriptions limited to the route or language chosen.

The standards process itself must adjust to the pace of change and be more adaptable. In fact, standards setting is a competitive effort to deliver solutions. The ability of the fleet-footed IETF (Internet Engineering Task Force) to deliver sufficient, even if over simple solutions, has given it an advantage over the slower moving organizations which either standardize the past or create inflexible standards for the unknown.

The IETF also has a further advantage of codifying practice rather than prescribing practice. X.400 embodied many assumptions about how email should work whereas the primitive protocols of the Internet, SMTP (Simple Mail Transport Protocol) coupled with domain naming, worked. Some of the limitations were later address with MIME encoding. This might be kludge layered upon kludge but it got the job done.

There is now fear that the IETF itself has become too laden with its own history.

Social Systems

Cooperating independent systems are, in effect, social systems. The participants try to interact by following rules that benefit them all. Norms are ideally derived by consensus after experience. Ultimately, each individual must take some of the responsibility for its own behavior and fate.

The Web is a good example. It spans continents and travels through (near) space. And it normally works!

The reason that it works is the unreliability of the worldwide Internet. Each element of the web is constructed on

the assumption that it will encounter errors and surprises over which it has no control. The Web, despite its spectacular growth and the vast number of services built on top of its protocols is simple and shallow. Simple in the sense that its architecture is understandable (so far) and shallow in that it isn't built upon layer upon layer but only using a simple transport protocol over a fairly robust connection protocol (HTTP on top of TCP/IP with HTML as a simple markup language).

There is a normal progression from the simple and viable to the complex and fragile that endangers the very robustness of the web. But it's core functionality will sustain it. The Web has also invigorated the Internet as a medium. The increased market size of the Internet due to the Web has spawned efforts such as Internet telephony. The Web is built upon the Internet, but the Internet is not limited to Web protocols.

This meandering path of change and innovation leaves us without a simple rule when we want to go in a specific direction and are not willing to wait passively for accidents to deliver solutions.

The basic principle of building a system out of cooperating elements applies. The key is to preserve robustness by having each element not just preserve reliability but regenerate it. Ideally it should be possible to recompose capabilities among their elements and subelements. This is difficult in arms-length relationships, though even there, negotiation should be part of the normal interaction, at least at the design level.

But what is most important is the attitude from which the problem is approached. Change and surprise are the normal state. To expect one design to continue to work is naive and dangerous. The term "bit rot" describes the process by which a seemingly stable set of bits or programs degenerates over time. The bits don't change but their relationship with their environment changes because of normal drift.

The nature of the problems we are trying to solve has changed and so have the tools. Rather than writing programs, people are specifying rules or policies or local behavior as part of their normal interactions with the world. Yet we don't understand how to compose these into coherent and explicable systems. We have had to learn the pragmatic debugging when theoretical debugging was proven impossible. Now we need to learn the heuristics that apply to compositing systems. To the extent that a system of these local rules can be observed and "understood", we can assist the user in understanding the behavior of the system.

This understanding needn't be full, just sufficient for to manage the interactions. In fact, we should expect that our understanding is incomplete and wrong so we can adapt to surprises.

In general, a descriptive approach to specifying behavior is much better than an algorithmic one since it allows for an overseer to "understand" what is being requested whereas a procedural description must be evaluated step by step without an overview. One is able to be descriptive precisely because the procedural elements are so well understood that they are incorporated into the larger system and we are simply calling upon these known elements of behavior. So we are back again to this cycle of building upon our previous understanding. The rate at which this process can propagate and iterate limits the rate of progress.

Towards Resiliency

The old goal was bug-free. The new goal is resiliency. It is much more important to recover from exceptions than to avoid them. The term "bug" is useful in describing a behavior in the purview of a single designer or design team. Failure to respond to the external failure or even simply the surprising behavior of another element is really different than a bug within one's own program.

This requires a shift in our thinking: from techniques for building programs to the integration of independent and partially defined elements. The programs are still there but so are other forms of specification such as policies and constraints. This is an environment in which surprises (AKA exceptions or failures) will be the norm. Since this is the infrastructure, there is no option for a complete "reboot", though local resets are allowed.

This resiliency applies as well to the "programs" by people providing specifications for behavior.

We are learning how to build resilient systems. The model of social systems provides some clues. At each scale there are organizations which define the limits the interactions with other organizations. These organizations also have mechanisms for regenerating local reliability. This is one of the tenets of the American federal government.

Conclusion

The world is full of limits imposed by physics and by the complexities of interacting chaotic systems. With cleverness and with the computer as tool for effecting computation we've pushed against these limits with great success. We'll continue to create faster and better systems.

The challenge now is to shift our thinking from improving systems in isolation to how create an infrastructure of interacting intelligent elements. We need to move from the goal of precise, bug-free but isolated, systems to resilient cooperating systems. Naively extending the rigid models

of computing will fail. But we can apply the lessons from earlier models of computing as long as we think them for this new arena. The solutions will be as much discovery as design. And we need to realize that it has always been so.

The rapid changes we've seen in computing systems have been due to our ability to quickly create tools that extend our capabilities. We've now gotten to the point where these changes are no longer confined to isolated systems.

The growth we've seen in the capabilities of computer systems is indeed remarkable but it can also be seen as the expected continuation of the hyper growth that occurs whenever there is a strong positive feedback cycle.

This feedback cycle is fed by the computer's ability to serve as a tool in its own advancement. It is also a means to leverage innovation from a large number of people.

Classic hypergrowth phenomena run out of resources or simply reach an unsupportable scale. We've avoided these limitations so far because the systems are becoming increasingly-resource efficient. We've been able to scale the systems because we've been able to regenerate reliability at each level.

This isn't always the case. We've been very unsuccessful in mastering the complexities of interactive systems and these complexities continue to increase as we interconnect systems and add intelligent elements throughout the systems.

We are not limited in what we can do with the systems. Innovation will continue to surprise us, the Web and Internet Telephony being but two of the most recent examples.

The Malthusians are very aware of the problems and challenges they confront. It is not reasonable to simply accept the premise that things will get better. There is no certainty and the advancements may be disconcerting to those looking for answers in terms of the current circumstances.

But it is the very uncertainty and chaos which allows new ideas to vie for superiority. This can be disconcerting for those who are looking for obvious solutions. But it is an exciting environment for innovation.

System design in the connected chaotic world requires resiliency. And we must regenerate reliability by discovering new simplicities.

You can't always get what you want,
But if you try sometimes you just might find
You just might find
You'll get what you need

The Rolling Stones, 1969