To Fernando J. Corbato
for his work in organizing the concepts and leading the development of the general-purpose large-scale time-sharing and resource-sharing computer systems CTSS and MULTICS

On Building Systems That Will Fail

FERNANDO J. CORBATÓ

It is an honor and a pleasure to accept the Alan Turing Award. My own work has been on computer systems, and that will be my theme. The essence of systems is that they are integrating efforts, requiring broad knowledge of the problem area to be addressed, and the detailed knowledge required is rarely held by one person. Thus the work of systems is usually done by teams. Hence I am accepting this award on behalf of the many with whom I have worked as much as for myself. It is not practical to name all the individuals who contributed. Nevertheless, I would like to give special mention to Marjorie Daggett and Bob Daley for their parts in the birth of CTSS and to Bob Fano and the late Ted Glaser for their critical contributions to the development of the Multics System.

Let me turn now to the title of this talk: "On Building Systems That Will Fail." Of course the title I chose was a teaser. I considered and discarded some alternate titles: "On Building Messy Systems," but it seemed too frivolous and suggests there is no systematic approach. "On Mastering System Complexity" sounded like I have all the answers. The title that came closest, "On Building Systems that are likely to have Failures" did not have the nuance of inevitability that I wanted to suggest.

What I am really trying to address is the class of systems that for want of a better phrase, I will call "ambitious systems." It almost goes without saying that ambitious systems never quite work as expected. Things usually go wrong—sometimes in dramatic ways. And this leads me to my main thesis, namely, that the question to ask when designing such systems is not: "*if* something will go wrong, but *when* it will go wrong?"

Some Examples

Now, ambitious systems that fail are really much more common than we may realize. In fact in some circumstances we strive for them, revelling in the excitement of the unexpected. For example, let me remind you of our national sport of football. The whole object of the game is for each team to play at the limit of its abilities. Besides the sheer physical skill required, one has the strategic intricacies, the ability to audibilize, and the quickness to react to the unexpected—all a deep part of the game. Of course, occasionally one team approaches perfection, all the plays work, and the game becomes dull.

Another example of a system that is too ambitious for perfection is military warfare. The same elements are there with opposing sides having to constantly improvise and deal with the unexpected. In fact we get from the military that wonderful acronym, SNAFU, which is politely translated as "situation normal, all fouled up." And if any of you are still doubtful, consider how rapidly the phrases "precision bombing" and "surgical strikes" are replaced by "the fog of war" and "casualties from friendly fire" as soon as hostilities begin.

On a somewhat more whimsical note, let me offer driving in Boston as an example of systems that *will* fail. Automobile traffic is an excellent case of distributed control with a common set of protocols called traffic regulations. The Boston area is notorious for the free interpretations drivers make of these pesky regulations, and perhaps the epitome of it occurs in the arena of the traffic rotary. A case can be made for rotaries. They are efficient. There is no need to wait for sluggish traffic signals. They are direct. And they offer great opportunities for creative improvisation, thereby adding zest to the sport of driving.

One of the most effective strategies is for a driver approaching a rotary to rigidly fix his or her head, staring forward, of course, secretly using peripheral vision to the limit. It is even more effective if the driver on entering the rotary, speeds up, and some drivers embellish this last step by adopting a look of maniacal glee. The effect is, of course, one of intimidation, and a pecking order quickly develops.

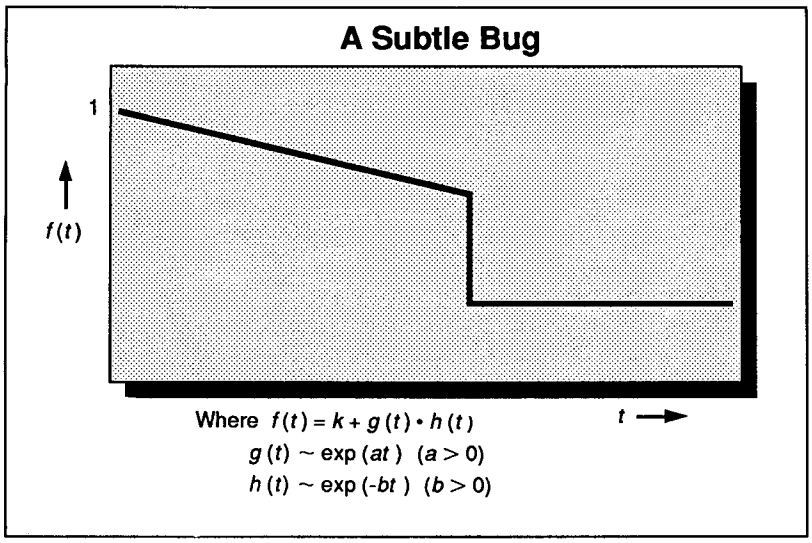
The only reason there are not more accidents is that most drivers have a second component to the strategy, namely, they assume everyone else may be crazy—they are often correct—and every driver is really prepared to stop with inches to spare. Again we see an example of a system where ambitious tactics and prudent caution lead to an effective solution.

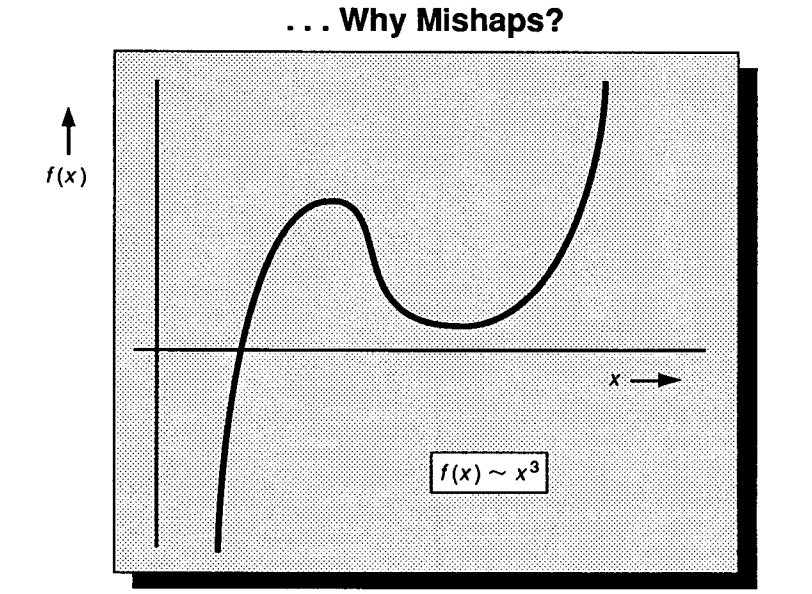
So far, the examples I have given may suggest that failures of ambitious systems come from the human element and that at least the technical parts of the system can be built correctly. In particular, turning to computer systems, it is only a matter of getting the code debugged. Some assume rigorous testing will do the job. Some put their hopes in proving program correctness. But unfortunately, there are many cases for which none of these techniques will always work [1]. Let me offer a modest example illustrated in Figure 1.

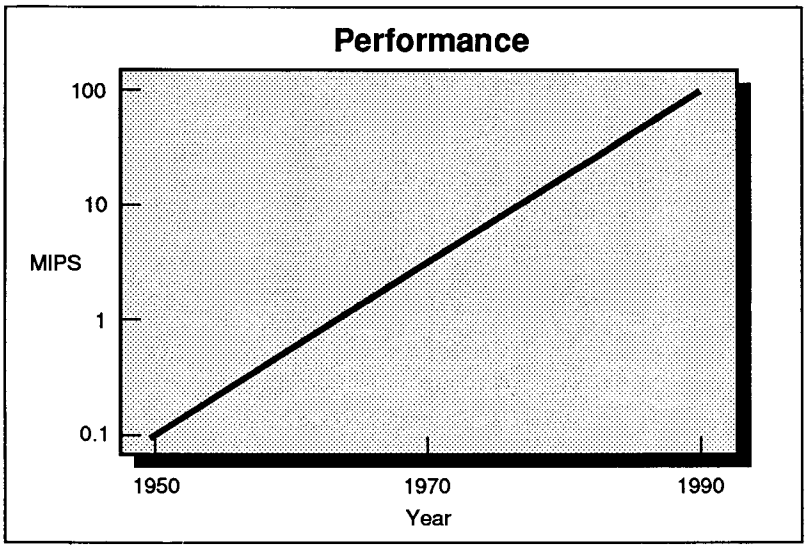
Consider the case of an elaborate numerical calculation with a variable, f, representing some physical value, being calculated for a set of points over a range of a parameter, t. Now the property of physical variables is that they normally do not exhibit abrupt changes or discontinuities.

So what has happened here? If we look at the expression for f, we see it is the result of a constant, k, added to the product of two other functions, g and h. Looking further, we see that the function g has a behavior that is exponentially increasing with t. The function h, on the other hand, is exponentially decreasing with t. The resultant product of g and h is almost constant with increasing t until an abrupt jump occurs and the curve for f goes flat.

What has gone wrong? The answer is that there has been floating-point underflow at the critical point in the curve, i.e., the representation of the negative exponent has exceeded the field size in the floating-







point representation for this particular computer, and the hardware has automatically set the value for the function h to zero. Often this is reasonable since small numbers are correctly approximated by zero—but not in this case, where our results are grossly wrong. Worse yet, since the computation of f might be internal, it is easy to imagine that the failure shown here would not be noticed.

Because correctly handling the pathology that this example represents is an extra engineering bother, it should not be surprising that the problem of underflow is frequently ignored. But the larger lesson to be learned from this example is that subtle mistakes are very difficult to avoid and to some extent are inevitable.

I encountered my next example when I was a graduate student programming on the pioneering Whirlwind computer. One night while awaiting my turn to use it, the graduate student before me began complaining of how "tough" some of his calculations were. He said he was computing the vibrational frequencies of a particular wing structure for a series of cases. In fact, his equations were cubics, and he was using the iterative Newton-Raphson method. For reasons he did not understand, his method was finding one of the roots, but not "converging" for the others. He was trying to fix this situation by changing his program so that when he encountered one of these tough roots, the program would abandon the iteration after a fixed number of tries.

Now there were several things wrong: First, the coefficients to his cubic equations were based on ex-

FIGURE 1
Debugging the Code

FIGURE 2
Nonconverging Iterative Method Caused by Poor Root Value

FIGURE 3
Performance of a Top-of-the-Line Computer by Decade

perimental data and some of his points were simply bad. Therefore, as Figure 2 illustrates, he only had one real root and a pair of imaginaries. Thus his iterative method could never converge for the second and third roots and the value of his first root was pure garbage. Second, cubic equations have an exact analytic closed form solution so that it was entirely unnecessary to use an iterative method. And third, based on his incomplete model and understanding of what was happening, he exercised very poor judgment in patching his program to ignore values that were seemingly difficult to compute.

Ambitious System Properties

Let me turn next to some of the general properties of ambitious systems. First, they are often vast and have significant organizational structures going beyond that of simple replication. Second, they are frequently complicated or elaborate and are too much for even a small group to develop. Third, if they really are ambitious, they are pushing the envelope of what people know how to do, and as a result there is always a level of uncertainty about when completion is possible. Because one has to be an optimist to begin an ambitious project, it is not surprising that underestimation of completion time is the norm. Fourth, ambitious systems when they work, often break new ground, offer new services and soon become indispensable. Finally, it is often the case that ambitious systems by virtue of having opened up a new domain of usage, invite a flood of improvements and changes.

Now one could argue that ambitious systems are really only difficult the first time or two. It is really only a matter of learning how to do it. Once one has, then one simply draws up the appropriate PERT charts, hires good managers, ensures an adequate budget and gets on with it. Perhaps there are some instances where this works, but at least in the area of computer systems, there is a fundamental reason it does not.

A key reason we cannot seem to get ambitious systems right is change. The computer field is intoxicated with change. We have seen galloping growth over a period of four decades and it still does not seem to be slowing down. The field is not mature yet and already it accounts for a significant percentage of the Gross National Product both directly and indirectly. More importantly the computer revolution—this second industrial revolution—has changed our life-styles and allowed the growth of countless new application areas. And all this change and growth not only has changed the world we live in, but has raised our expectations, spurring on increasingly ambitious systems in such diverse areas as airline reservations, banking, credit cards, and air traffic control to name only a few.

Behind the incredible growth of the computer industry is, of course, the equally mind-boggling change that has occurred in the raw performance of digital logic. Figure 3, which is not precise and which many of you have seen before in some form, gives the performance of a top-of-the-line computer by decade. The ordinate in MIPS is logarithmic as you can see. In particular in the last decade, the graph becomes problem dependent so that the upper right-hand end of the line should break up into some sort of whiskers as more and more computers are tailored for special applications and for parallelism.

Complicating matters too is that parallelism is not a solution for every problem. Certain calculations that are intrinsically serial, such as rocket trajectories, derive very limited benefit from parallel computers. And one of course is reminded of the old joke about the Army way of speeding up pregnancy by having nine women spend one month at the task.

As Figure 4 makes clear, it is not just performance that has fueled growth but rather cost/performance, or simply put, favorable economics. The graph is an oversimplification, but represents the cost for a given performance computer model over the last four decades. Again the ordinate is logarithmic, going from 10 million dollars in 1950 down to one thousand dollars in 1990. As we approach the present, corresponding to a personal computer, the graph really should become more complicated since one consequence of computers becoming super-cheap is that increasingly, they are being embedded in other equipment. The modern automobile is but one example. And it **remains to be seen how general-purpose the current wave of palm-sized computers will be with their stylus inputs.**

Further, when we look at a photograph taken around 1960 of a "machine room" staffed with one lone operator, we are reminded of the fantastic changes that have occurred in computer technology. The boxes are huge, shower-stall-sized, and the overall impression is of some small factory. You were supposed to be impressed and the operator was expected to maintain decorum by wearing a necktie. And if he did not, at least you could be sure an IBM maintenance engineer would.

Another reminder of the immense technological change which has occurred is in the physical dimensions of the main memories of computers. For example, if one looks at old photographs taken in the mid-1950s of core memory sys-

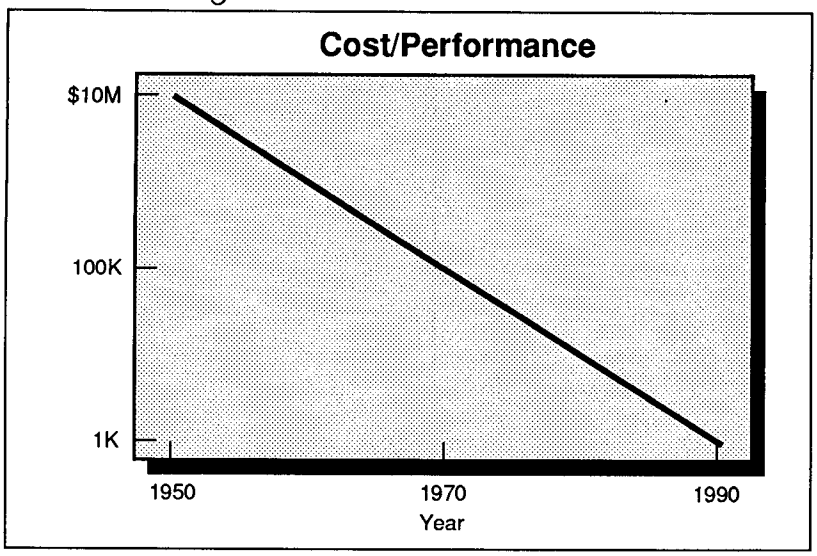


FIGURE 4
Cost for Given-Performance Computer Model over Four Decades

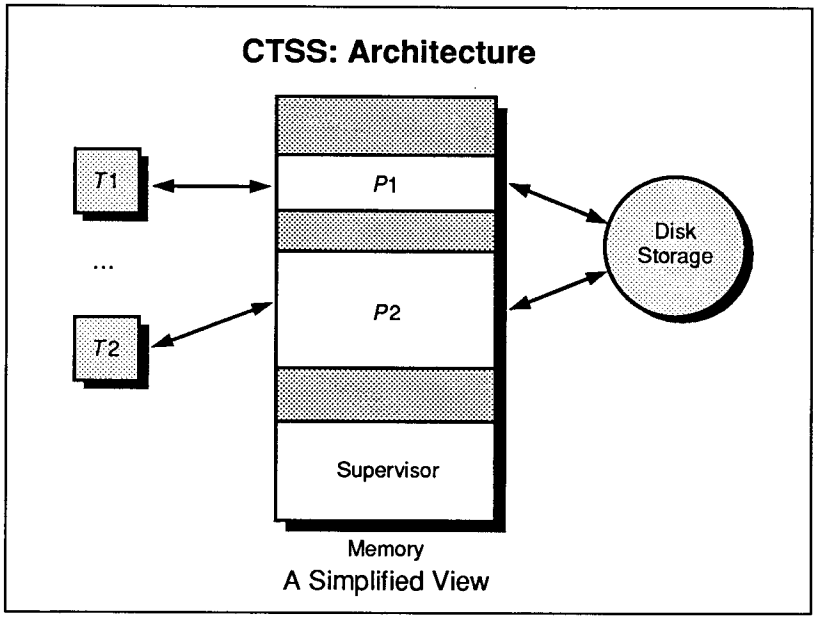


FIGURE 5
Input/Output of User Programs

tems, one typically sees a core memory plane roughly the size of a tennis racquet head which could hold about 1,000 bits of information. Contrast that with today's 4megabit memory chips that are smaller than one's thumb.

The basis of the Award today is largely for my work on two pioneering time-sharing systems, CTSS [5, 6] and Multics [7, 9]. Indeed, it is from my involvement with those two systems that I gained the system-building perspective I am offering. It therefore seems appropriate to take a brief retrospective look at these two systems as examples of ambitious systems and to explore the reasons why the complexity of the tasks involved made it almost impossible to build the systems correctly the first time [2].

CTSS, The Compatible Time-Sharing System

Looking first at CTSS, let us remember the dark ages that existed then. This was the early 1960s. The computers of the day were big and expensive, and the administrators of computing centers felt obliged to husband the precious resource. Users, i.e., programmers, were expected to submit a computing job as a deck of punched cards. These were then combined into a batch with other jobs onto a magnetic tape and the tape was processed by the computer operating system. It had all the glamour and excitement of dropping one's clothes off at a laundromat.

The problem was that even for a trivial input typing mistake, the job would be aborted. Time-sharing, as most of you know, was the solution to the problem of not being able to interact with computers. The general vision of modern time-sharing was primarily spelled out by John McCarthy, who I am pleased to note is a featured speaker at this conference. In England, Christopher Strachey independently came up with a limited kind of interactive computing, but it was aimed mostly at debugging. Soon there were many groups around the country developing various forms of interactive computing, but in almost all cases, the resulting systems had significant limitations.

It was in this context that my own group developed our version of the time-sharing vision. We called it The Compatible Time-Sharing System, or CTSS for short. Our initial aspirations were modest. First, the system was meant to be a demonstration prototype before more ambitious designs being attempted by others could be implemented. Second, it was intended to handle general-purpose programming. And third, it was meant to make it possible to run most of the large body of software that had been developed over the years in the batch-processing environment. Hence the name.

The basic scheme used to run

CTSS was simple. The supervisor program, which was always in main memory, would commutate among the user programs, running each in turn for a brief interval with the help of an interval timer. As Figure 5 indicates, user programs could do input/output with the typewriter-like terminals and with the disk storage unit as well.

But the diagram is oversimplified. The key difficulty was that main memory was in short supply and not all the programs of the active users could remain in memory at once. Thus the supervisor program not only had to move programs to and from the disk storage unit, but it also had to act as an intermediary for all I/O initiated by user programs. Thus all the I/O lines should only point to the supervisor program.

As a further complication, the supervisor program had to prevent user programs from trampling over one another. To do this required special hardware modifications to the processor such that there were memory bound registers that could only be set by the supervisor. Nevertheless, despite all the complications, the simplicity of the initial supervisor program allowed it to occupy about 22 Kbytes of storageless storage than required for the text of this talk!

Most of the battles of creating CTSS involved solving problems which at the time did not have standard solutions. For example: There were no standard terminals. There were no simple modems. I/O to the computer was by word and not by character, and worse yet, did not accommodate lower case letters. The computers of the day had neither interrupt timers nor calendar clocks. There was no way to prevent user programs from issuing raw I/O instructions at random. There was no memory protection scheme. And, there was no easy way to store large amounts of data with relatively rapid random access.

The overall result of building CTSS was to change the style of computing, but there were several effects that seem worth noting. One of the most important was that we discovered that writing interactive software was quite different from software for batch operation and even today, in this era of personal computers, the evolution of interactive interfaces continues.

In retrospect, several design decisions contributed to the success of CTSS, but two were key. First, we could do general-purpose programming and, in particular, develop new supervisor software using the system itself. Second, by making the system able to accommodate older batch code, we inherited a wealth of older software ready-to-go.

One important consequence of developing CTSS was that for the first time, users had persistent on-line storage of programs and data. Suddenly the issues of privacy, protection and backup of information had to be faced. Another byproduct of the development was that because we operated terminals via modems, remote operation became the norm. Also, the new-found freedom of keeping information on-line in the central file system suddenly made it especially convenient for users to share and exchange information among themselves.

And there were surprises too. To our dismay, users who had been enduring several-hour waits between jobs run under batch processing were suddenly restless when response times were more than a second. Moreover, many of the simplifying assumptions that had allowed CTSS to be built so simply, such as a one-level file system, suddenly began to chafe. It seemed like the more we did, the more users wanted.

There are two other observations that can be made about the CTSS system. First, it lasted far longer than we expected. Although CTSS had been demonstrated in primitive form in November 1961, it was not until 1963 that it came into wide use as the vehicle of a Project MAC Summer Study. For a time there were two copies of the system hardware, but by 1973 the last copy was turned off and scrapped primarily because the maintenance costs of the IBM 7094 hardware had become prohibitively expensive, and up to the bitter end, there were users desperately trying to get in a few last hours of use.

Second, the then-new transistors and large random-access disk files were absolutely critical to the success of time-sharing. The previous generation of vacuum tubes was simply too unreliable for sustained real-time operation and, of course, large disk files were crucial for the central storage of user programs and data.

A Mishap

My central theme is to try to convince you that **when you have a multitude of novel issues to contend with while building a system, mistakes are inevitable.**

And indeed, we had a beauty while using CTSS. Let me describe it:

What happened was that one afternoon at Project MAC, where CTSS was being used as the main time-sharing workhorse, any user who logged in, found that instead of the usual message-of-the-day typing out on his or her terminal, he had the entire file of user passwords. This went on for 15 or 20 minutes, until one particularly conscientious user called the system administrator and began the conversation with "Did you know that . . . ?" Needless to say, there was general consternation with this co-

lossal breach of security, the system was hastily shut down and the next twelve hours were spent heroically changing everyone's password. The question was how could this have happened? Let me explain.

To simplify the organization of the initial CTSS system, a design decision had been made to have each user at a terminal associated with his or her own directory of files. Moreover, the system itself was organized as a kind of quasi-user with its own directory that included a large number of supporting applications and files, including the message-of-the day and the password file. So far, so good. Normally a single-system programmer could login to the system directory and make any necessary changes. But the number of system programmers had grown to about a dozen in number, and, further, the system by then was being operated almost continuously so that the need to do live maintenance of the system files became essential. Not surprisingly, the system programmers saw the one-user-to-a-directory restriction as a big bottleneck for themselves. They thereupon proceeded to cajole me into letting the system directory be an exception so that more than one person at a time could be logged into it. They assured me that they would be careful to not make mistakes.

But of course a mistake was made. A software design decision in the standard system text editor was overlooked. It was assumed that the editor would only be used by one user at a time working in one directory so that a temporary file could have the same name for all instantiations of the editor. But with two system programmers editing at the same time in the system directory, the editor temporary files became swapped and the disaster occurred.

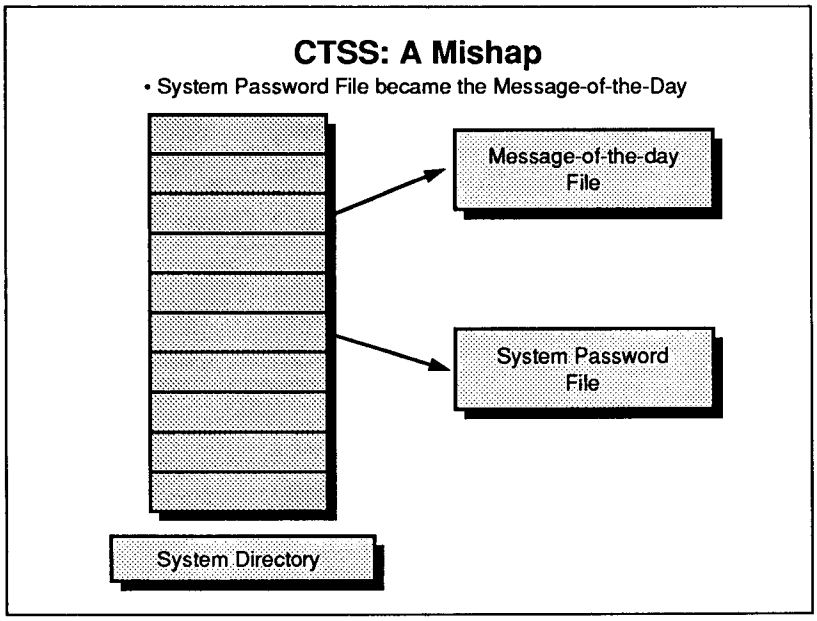


FIGURE 6
***CTSS* Is Full of Surprises**

One can draw two lessons from this: First, design bugs are often subtle and occur by evolution with early assumptions being forgotten as new features or uses are added to systems. Second, even skilled programmers make mistakes.

Multics

Let me turn now to the development of Multics [12]. I will be brief since the system has been documented well and there have already been two retrospective papers written [3, 4]. The Multics system was meant to do time-sharing "right" and replace the previous ad hoc systems such as CTSS. It started as a cooperative effort among Project MAC of MIT, the Bell Telephone Laboratories, and the Computer Department of General Electric, later acquired by Honeywell. In our expansiveness of purpose we took on a long list of innovations.

Among the most important ones were the following: First, we introduced into the processor hardware the mechanisms for paging and segmentation along with a careful scheme for access control. Second, we introduced an idea for rings of protection around the supervisor software. Third, we planned from the start that the system would be composed of interchangeable multiple processors, memory modules, and so forth. And fourth, we made the decision to implement nearly all of the system in the newly defined compiler language, PL/I.

Let me share a few of my observations about the Multics experience. The novel hardware we had commissioned meant that the system had to be built from the ground up so that we had an immense task on our hands.

The decision to use a compiler to implement the system software was a good one, but what we did not appreciate was that new language PL/I presented us with two big difficulties: First, the language had constructs in it which were intrinsically complicated, and it required a learning period on the part of system programmers to learn to avoid them. Second, no one knew how to do a good job of implementing the compiler. Eventually we overcame these difficulties but it took precious time.

That Multics succeeded is remarkable, for it was the result of a cooperative effort of three highly independent organizations and had no administrative head. This meant decisions were made by persuasion and consensus. Consequently, it was difficult to reject weak ideas until considerable time and effort had been spent on them.

The Multics system did turn into

a commercial product. Some of its major strengths were the virtual memory system, the file system, the attention to security, the ability to do online reconfiguration, and the information backup system for the file system.

And, as was also true with CTSS, many of the alumni of the Multics development have gone on to play important roles in the computing field [11].

A few more observations can be made about the ambitious Multics experience. In particular, we were misled by our earlier successes with previous systems such as CTSS, where we were able to build them "brick-by-brick," incrementally adding ideas to a large base of already working software.

We also were embarrassed by our inability to set and meet accurate schedules for completion of the different phases of the project. In retrospect, we should not have been, for we had never done anything like it before. However in many cases, our estimations should have been called guesses.

The Unix system [15] was a reaction to Multics. Even the name was a joke. Ken Thompson was part of the Bell Laboratories' Multics effort, and, frustrated with the attempts to bring a large system development under control, decided to start over. His strategy was clear—Start small and build up the ideas one by one as he saw how to implement them well. As we all know, Unix has evolved and become immensely successful as the system of choice for workstations. Still there are aspects of Multics that have never been replicated in Unix.

As a commercial product of Honeywell and Bull, Multics developed a loyal following. At the peak there were about 77 sites worldwide and even today many of the sites tenaciously continue for want of an alternative.

Sources of Complexity

The general problem with ambitious systems is complexity. Let me next try to abstract some of the major causes. The most obvious complexity problems arise from scale. In particular, the larger the personnel required, the more levels of management there will be. We can see the problem even if we use simplistic calculations. Thus if we assume a fixed supervision ratio, for example six, the levels of management will grow as the logarithm of the personnel. The difficulty is that with more layers of management, the top-most layers become out of touch with the relevant bottom issues and the likelihood of random serendipitous communication decreases.

Another problem of organizations is that subordinates hate to report bad news, sometimes for fear of "being shot as the messenger" and at other times because they may have a different set of goals than the upper management.

And finally, large projects encourage specialization so that few team members understand all of the project. Misunderstandings and miscommunication begin, and soon a significant part of the project resources are spent fighting internal confusion. And, of course, mistakes occur.

My next category of complexity arises because of new design domains. The most vivid examples come from the world of physical systems, but software too is subject to the same problems, albeit often in more subtle ways.

Consider the destruction of the Tacoma Narrows Bridge, in Washington State, on November 7, 1940. The bridge had been proudly opened about four months earlier. Many of you have probably seen the amateur movie that was fortunately made of the collapse. What happened is that a strong but not unusual crosswind blew that day. Soon the roadbed, suspended by cables from the main span, began to vibrate like a reed, and the more it flexed, the better cross section it presented to the wind. The result was that the bridge tore itself apart as the oscillations became large and violent. What we had was a case of a new design domain where the classic bridge builder, concerned with gravity-loaded structures, had entered into the realm of aeronautics. The result was a major mistake.

Next, let us look at the complexities that arise from human usage of computer systems. In using online systems that allow the sharing or exchanging of information—and here networked workstations clearly fall in this class—one is faced with a dilemma: If one places total trust in all other users, one is vulnerable to the antisocial behavior of any malicious user—consider the case of viruses.

But if one tries to be totally reclusive and isolated, one is not only bored, but one's information universe will cease to grow and be enhanced by interaction with others.

The result is that most of us operate in a complicated trade-off zone with various arrangements of trust and security mechanisms. Even such simple ideas as passwords are often a problem. They are a nuisance to remember, they can easily be compromised inadvertently, and they cannot be selectively revoked if shared. Privacy and security issues

are particularly difficult to deal with since responsibilities are often split among users, managers, and vendors.

Worse yet, there is no way to simply "look" at a system and determine what the privacy and security implications are. It is no wonder mistakes occur all the time in this area.

One of the consequences of using computer systems is that increasingly information is being kept on-line in central storage devices. Computer storage devices have become remarkably reliable—except when they break—and that is the rub. Even the most experienced computer user can find him- or herself lulled into a false sense of security by the almost perfect operation of today's devices. The problem is compounded by the attitude of vendors, not unlike the initial attitude of the automobile industry toward safety, where inevitable disk failure is treated as a negative issue that dampens sales.

What is needed is constant vigilance against a long list of "what ifs": hardware failure, human slips, vandalism, theft, fire, earthquakes, long-term media failure, and even the loss of institutional memories concerning recovery procedures. And as long as some individuals have to "learn the hard way," mistakes will continue to be made.

A further complication in discussing risk or reliability is that there is not a good language with which to carry on a dialog. Statistics are as often misapplied as they are misunderstood. We also get absurd absolutes such as "the Strategic Defense Initiative will produce a perfect unsaturatable shield against nuclear attack" [14] or "it is impossible for the reactor to overheat." The problem is that we always have had risks in our lives, we never have been very good at discussing them, and with computers we now have a lot of new sources.

Another source of complexity arises with rapid change, change which is often driven by technology improvements. A result is that changes in procedures or usage occur and new vulnerabilities can arise. For example, in the area of telephone networks, the economies and efficiencies of fiber optic cables compared to copper wire are rapidly causing major upgrades and replacements in the national telephone plant. Because one fiber cable can carry at a reasonable cost the equivalent traffic of thousands of copper wires, fiber is quickly replacing copper. As a result, a transformation is likely to occur where network links become sparser over a given area and multiply interconnected nodes become less connected.

The difficulty is that there is reduced redundancy and a much higher vulnerability to isolated accidents. In the Chicago area not long ago there was a fire at a fiber optics switching center that caused a loss of service to a huge number of customers for several weeks. More recently, in New York City there was a shutdown of the financial exchanges for several hours because of a single mishap with a backhoe in New Jersey. Obviously in both instances, efficiency had gotten ahead of robustness.

The last source of complexity that I will single out arises from the frailty of human users when forced to deal with the multiplicity of technologies in modern life. In a little more than a century, there has been an awesome progression of technological changes from telephones and electricity, through automobiles, movies and radio—I will not even try to complete the list since we all know it well. The overall consequence has been to produce vast changes in our life-styles, and we see these changes even happening today. Consider the changes in the television editing styles that have occurred over a few decades, the impact of viewgraph overhead projectors on college classrooms, and the way we now do our banking with automatic teller machines. And the progression of life-style changes continues at a seemingly more rapid pace with word processing, answering machines, facsimile machines, and electronic mail.

One consequence of the many life-style changes is that some individuals feel stressed and overstimulated by the plethora of inputs. The natural defense is to increasingly depend on others to act as information filters. But the combination of stressful life-styles and insulation from original data will inevitably lead to more confusion and mistakes.

Conclusions

Most of this talk has been directed toward trying to persuade you that failures in complex, ambitious systems are inevitable. However, I would be remiss if I did not address ways to resolve the problem. Unfortunately, the list I can offer is rather short but worthy of brief review.

First, it is important to emphasize the value of simplicity and elegance, for complexity has a way of compounding difficulties and as we have seen, creating mistakes. My definition of elegance is the achievement of a given functionality with a minimum of mechanism and a maximum of clarity.

Second, the value of metaphors should not be underestimated. Metaphors have the virtue of an expected behavior that is understood by all. Unnecessary communication and misunderstandings are reduced. Learning and education are quicker. In effect, metaphors are a way of internalizing and abstracting concepts allowing one's thinking to be on a higher plane and low-level mistakes to be avoided.

Third, use of constrained languages for design or synthesis is a powerful methodology. By not allowing a programmer or designer to express irrelevant ideas, the domain of possible errors becomes far more limited.

Fourth, one must try to anticipate both errors of human usage and of hardware failure and properly develop the necessary contingency paths. This process of playing "what if" is not as easy as it may sound, since the need to attach likelihoods of occurrence to events and to address issues of the independence of failures is implicit.

Fifth, it should be assumed in the design of a system, that it will have to be repaired or modified. The overall effect will be a much more robust system, where there is a high degree of functional modularity and structure, and repairs can be made easily.

Sixth, and last, on a large project, one of the best investments that can be made is the cross education of the team so that nearly everyone knows more than he or she needs to know. Clearly, with educational redundancy, the team is more resilient to unexpected tragedies or departures. But in addition, the increased awareness of team members can help catch global or systemic mistakes early. It really is a case of "more heads are better than one."

Finally, I have touched on many different themes in this talk but I will single out three: First, the evolution of technology supports a rich future for ambitious visions and dreams that will inevitably involve complex systems. Second, one must always try to learn from past mistakes, but at the same time be alert to the possibility that new circumstances require new solutions. And third, one must remember that ambitious systems demand a defensive philosophy of design and implementation. In other words, "Don't wonder *if* some mishap may happen, but rather ask *what* one will do about it when it does occur." ◻

References

1. Brooks, F.P., Jr. No silver bullet. *IEEE Comput.* (Apr. 1987), 10–19.
2. Corbató, F.J. Sensitive issues in the design of multi-use systems. Unpublished lecture transcription of Feb. 1968, Project MAC Memo M-383.
3. Corbató, F.J., and Clingen, C.T. A managerial view of the Multics system development. In *Research Directions in Software Technology,* P. Wegner, Ed., M.I.T. Press, 1979. (Also published in *Tutorial: Software Management,* D.J. Reifer, Ed., IEEE Computer Society Press, 1979; Second Ed., 1981; Third Ed., 1986.)
4. Corbató, F.J., Clingen, C.T., and Saltzer, J.H. Multics: The first seven years. In *Proceedings of the SJCC* (May 1972), pp. 571–583.
5. Corbató, F.J., Daggett, M.M., and Daley, R.C. An experimental time-sharing system. In *Proceedings of the Spring Joint Computer Conference* (May 1962).
6. Corbató, F.J., Daggett, M.M., Daley, R.C., Creasy, R.J., Hellwig, J.D., Orenstein, R.H., and Horn, L.K. *The Compatible Time-Sharing System: A Programmer's Guide.* M.I.T. Press, June 1963.
7. Corbató, F.J., and Vyssotsky, V.A. Introduction and overview of the Multics system. In *Proceedings FJCC* (1965).
8. Daley, R.C. and Neumann, P.G. A general-purpose file system for secondary storage. In *Proceedings FJCC* (1965).
9. David, E.E., Jr. and Fano, R.M. Some thoughts about the social implications of accessible computing. In *Proceedings FJCC* (1965).
10. Glaser, E.L., Couleur, J.F. and Oliver, G.A. System design of a computer for time-sharing applications. In *Proceedings FJCC* (1965).
11. Neumann, P.G., a Multics veteran, has become a major contributor to the literature of computer-related risks. He is the editor of the widely-read network magazine "Risks-Forum," writes the "Inside Risks" column for the CACM, and periodically creates digests in the ACM Software Engineering Notes.
12. Organick, E.I. *The Multics System: An Examination of its Structure.* MIT Press, 1972.
13. Ossanna, J.F., Mikus, L. and Dunten, S.D. Communications and input-output switching in a Multiplex computing system. In *Proceedings FJCC* (1965).
14. Parnas, D.L. Software aspects of strategic defense systems. *Am. Sci.* (Nov. 1985). An excellent critique on the difficulties of producing software for large-scale systems.
15. Ritchie, D.M. and Thompson, K. The UNIX time-sharing system. *Commun. ACM 17,* 7 (July 1974), 365–375.
16. Vyssotsky, V.A., and Corbató, F.J. Structure of the Multics Supervisor. In *Proceedings FJCC,* 1965.

CR Categories and Subject Descriptors: C.2 [Computer Systems Organization]: Computer-Communication Networks; C.4 **[Computer Systems Organization]:** Performance of Systems; D.4 **[Software]:** Operating Systems; H.5 **[Information Systems]:** Information Interfaces and Presentation; K.2 **[Computing Milieux]:** History of Computing

General Terms: Design

About the Author:

FERNANDO J. CORBATÓ is Associate Head of Computer Science and Engineering at the Massachusetts Institute of Technology. **Author's Present Address:** Computer Science and Engineering Department, Room NE43-514, 545 Technology Square, Cambridge, MA 02139.