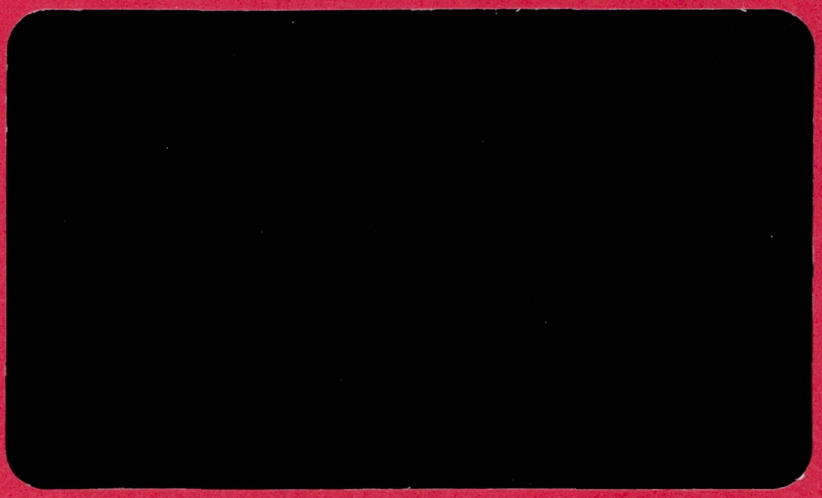

TECHNICAL REPORT

Department of Computer Science
Cornell University
Ithaca, New York

How Robust are Distributed Systems?

K. P. Birman*

TR 89-1014
June 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037, Contract N0014-87-C-8904, and also by a grant from the Siemens Corporation. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense Analysis position, policy, or decision.

How Robust are Distributed Systems?

K. P. Birman

Department of Computer Science
Cornell University
Ithaca, NY 14853

This is a preprint of material that will appear in the collected lecture notes from *Arctic '88, An Advanced Course on Operating Systems*, Tromso, Norway, July 5-14, 1988. The lecture notes will appear in book form later this year.

---

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 6037, Contract N0014-87-C-8904, and also by a grant from the Siemens Corporation. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense Analysis position, policy, or decision.

20

How robust are distributed systems?

K. P. Birman

I started writing this chapter in November 1988, shortly after a "worm" was unleashed in the internet; by exploiting network security loopholes it penetrated and crashed large numbers of machines.[1] Coincidentally, newspapers were filled with retrospective analyses of the 1987 stock market crash.[2] Both events gave rise to speculation concerning the robustness of contemporary distributed systems, and it is to this topic that I address myself.

Before beginning, it is important to recognize that these episodes also touch on rather deep ethical questions. One can and should ask about the propriety of writing and running a program that has no constructive purpose, or even of pitting small investors against massive institutions armed with supercomputers.

Personally, I feel that the running a worm shows a deplorable lack of judgement, and entertain some doubts about the modern stock market. Nonetheless, these conclusions are debatable, and strongly dependent on questions of taste. The present discussion focuses on a more technical issue, namely the robustness of distributed computing systems — against intrusions, but also in the presense of events that arise commonly in distributed settings, such as failures and overloads. Because these issues are basically technical, one can hope to arrive at a more or less technical answers to them. To the extent that these lead back to philosophical speculations, the questions raised concern implications of more technical conclusions, and hence one might hope that they will be less controvertial than

[1] The program was designed to penetrate as many machines as possible using bugs and loopholes in UNIX communication and mail-handling software. Although apparently intended to unobtrusively maintain a low level of "infection," a programming error caused the worm to replicate much faster than intended. It gained access to nearly 6000 systems during a 48-hour period, overloading and crashing a large percentage.

[2] This is in reference to the dramatic stock market declines that occurred during a world-wide flurry of program-driven trading in October of 1987.

conclusions arrived at using, for example, ethical principles that might not be universally accepted.

20.1. Predicting the behavior of a distributed system

Consider the problem of predicting how a distributed system will behave while it is executing. Such a system will be made up of large numbers of components, operating asynchronously from one another and hence with incomplete and inaccurate views of one-another's state. Moreover, few distributed systems operate in a steady state: load fluctuations are common as new tasks arrive and active tasks terminate. Jointly, these aspects make it nearly impossible to arrive at detailed predictions.

For example, feedback can arise in an automated stock trading system because programmed trade decisions are based on market indexes that change rapidly to reflect recent trading. If all trading programs operate independently, this feedback effect is minimal. However, if a condition provokes sell decisions in large numbers of programs, or exceptionally large sell orders, it can reinforce itself by driving those indexes down, triggering waves of sales. Such a sequence apparently led to the 1987 crash. Whether or not one questions the use of trading systems in general, it seems obvious that one could question the use of trading programs subject to such behavior. What is less obvious is that these sorts of behaviors are unpredictable and can arise from seemingly trivial mechanisms.

A behavior prediction problem also arose as an issue in the 1988 worm incident. One way to design a worm would be to write a distributed protocol that maintains a replicated list of currently infected sites, by having worm programs communicate directly with one another and monitor one another's status to detect failures. Using this approach, one could maintain a very stable population of worms, infecting new sites in a highly controlled manner. However, the protocol would be hard to design — similar problems were discussed in Chapters 14 and 15. An easier problem is to implement such an algorithm given atomic group addressing and broadcast primitives, but the designer of a worm cannot (yet) assume that such primitives are available.

In an ill-fated decision, the designer of the 1988 worm evidently turned instead to a random algorithm. Under this approach, each worm independently makes decisions to infect neighboring sites based on probabilistic mechanisms. The resulting worm population is influenced by factors that include the current population, the rate of new infections, the death rate, and the probability of a successful penetration of a system. For certain values of these parameters, the worm population might well remain stable and small. However, for other values, an *unstable* solution results, whereby the worm population will die out or grow uncontrollably. The question is thus how to pick parameter values that will definitely give stable populations. Unfortunately for the designer of the worm, problems of this sort are often intractable, and this one almost certainly is. Current mathematics gives little insight into how one might pick the

parameters to ensure stability, or even test for stability given particular choices of parameters. The 1988 worm thus had an intrinisic and probably insurmountable flaw.

It is striking that whereas the worm provoked much discussion of distributed systems security, and some attention was been given to the ethical implications of running such a program, rather little was paid to the broader issue of which the worm was just a manifestation. Many systems contain feedback mechanisms, for example in schedulers and in the flow-control mechanisms used by the communication layer. There is growing interest in applying these sorts of systems in a wide range of critical settings. How can one be sure that a given system is secure and immune to chaotic behavior? Lacking this knowledge, should one not expect other such incidents, perhaps with catastrophic consequences?

Until recently a laboratory rarity, distributed systems have become pervasive, and our society has come to rely on them over a five-year period. Increasingly, systems such as these replace humans who cannot provide the sorts of predictable realtime responsiveness of a computer. Yet, as these episodes illustrate, however bright the *promise* of distributed computing, the technology is also associated with significant risks.

20.2. Technology and social responsibility

I believe that the inventors of a technology assume an obligation to overcome flaws in that technology, especially flaws that could exact a direct human cost. Too many technologies have been turned loose without adequate consideration of where they might lead. The more critical a technology, the more important that its weaknesses be anticipated before they become stumbling blocks. To fail to confront this issue in the context of distributed computer systems invites haphazard interconnection of machines using mechanisms capable of interacting in unanticipated ways. Lacking explicit actions to the contrary, one must anticipate that confidential data will be increasingly often exposed to intrusions, that critical control facilities will increasingly often be subject to disruption, and that failures of all sorts will be increasingly common.

This argument can be carried even further. In many cases, sober analysis leads to the realization that a technology simply cannot be perfected to the degree needed in the time available, if ever. A good example, strongly dependent on distributed computing technologies, is launch-on-warning software for controlling the nation's strategic weapons systems. These systems have been proposed because human beings cannot function rapidly enough to make launch decisions in response to a surprise attack. Unfortunately, the proponents of new weapons technologies have often overlooked weaknesses of a technology, and the limits on the degree to which it can be perfected. Can one really build a large distributed system that is sufficiently robust to entrust it to perform such a crtical task? Based on the arguments that I will advance below, I think the answer is a negative one. It seems to me that there is an applicable "impossibility" result;

every bit as serious a limitation as any theoretically provable one. And, similar arguments seem to apply in many other settings. To establish this, however, one must first ask how robust a distributed system can reasonably be expected to be.

In the case of more mature technologies, such as transportation and power generation, organizations exist to ensure the safety of systems that enter widespread use. The measures mandated in some areas are astonishing in their pessimism about human potential for error and for assuming that unlikely events will not only occur, but will do so at the worst possible time. For example, nuclear reactors incorporate the most extreme measures to minimize risk. This has clearly reduced the potential for disaster. Yet, incidents continue to occur, and in many cases the ways in which they occur raise new questions about the whole assumption that systems of this sort can ever be made safe.

In contrast, the engineering of even the most widely used distributed systems has been fairly informal. If trains crash and nuclear "excursions" (leaks) occur despite every countermeasure that designers with years of experience have managed to devise, should one not expect frequent disruptions in distributed systems designed with only minimal attention to robustness? The most common form of regulation for distributed systems has been through low-level standards, as for the ISO data transport protocols. However, the problems identified above arise at the application level, and to the extent that applications-level standards have been developed, they have been premature and overly restrictive. Clearly, one cannot define a standard for aspects of a system that are still experimental. Yet, it seems equally clear that ignoring these issues only encourages the construction of complex, fragile software.

20.3. Principles for distributed computing

One thing that we lack is a set of guiding principles to encourage the development of sound solutions to distributed computing problems. Let me propose a set of such principles now.

Assume responsibility.

Those who produce distributed computing software should make every effort to ensure that the software is safe for its intended mode of use and that it can only be used in the intended way. And, we must accept our responsibility to apply the highest standards of ethical behavior in our individual research and to instill these standards in our students and colleagues.

Interconnect for good reasons.

Systems should be interconnected to achieve concrete objectives, not in the abstract belief that interconnection is a good thing. Systems that are incapable of interacting are incapable of compromising one another.

Support only necessary services.

When systems are interconnected, the default should be to support the smallest possible set of services. Services should be enabled selectively and because there is a good reason to support them. This minimizes the probability that a loophole in the large (and ever increasing) set of communication services could have widespread consequences. Also, it makes it more likely that the services that are enabled will be properly maintained.

This is especially important for services implemented anonymously and provided as executables (without source). For example, the 1988 worm made use of a bugs in the UNIX remote finger and mail handling utilities. One might ask just what purpose was served by enabling these on the majority of the machines that were compromised. Many users maintain a primary account on just one of the machines with which they work, and neither receive mail nor maintain finger databases on other machines. Many machines, in fact, are used in ways that preclude reception of mail or finger queries. Yet, the default has been to enable every possible service whether needed or not, and substantial expertise is often required to selectively *disable* an unwanted service. This pursuit of uniformity and flexibility has had a paradoxical outcome: resources are consumed to run services that are not useful, and the machines on which they run are made less robust.

Include self-diagnosis and authentication mechanisms

When communication is permitted and a service is supported, authenticate the origin and legality of requests. Many current networks make "punning" (misrepresentation of origin information) too easy, giving the illusion of security where there is actually none.

Authentication is an issue beyond its security implications. It is widely accepted that procedures should authenticate their arguments. Large distributed systems should carry this principle further. Mechanisms are needed by which whole system components can monitor themselves continuously, actively looking for inconsistencies and shutting themselves down if problems are detected. The reasoning here is that although software bugs may be inevitable, if they are detected rapidly the consequences can often be limited, for example by explicitly halting and restarting affected programs. This approach has long been used successfully in electronic circuit switching.

Design for fault-tolerance.

Far too many distributed systems are designed as if failures will not occur, or give undefined behavior in the presense of failures. This is precisely the converse of the attitude needed when building software to survive a wide range of communication and hardware disruptions, especially in light of the self-checking mechanism proposed above. To build a robust distributed system, one must assume that failures will occur. The choice is to try to survive such events, or to

detect them and shut down before an inconsistent or erroneous action could result.

What faults should be treated? It is generally agreed that human behavior will violate any rules one attempts to impose. Thus, the traditional approach in systems that must interact with humans is to design for tolerance of the largest conceivable class of behaviors. In contrast, designers of distributed systems generally assume independent, benign machine failures, and that communication failures involve only packet loss, duplication, unsequenced delivery or partitioning — not message corruption, forgery, or protocol violations.

Although one can question whether failures are always benign and independent, there are practical difficulties with using more demanding failure models. Most wider classes of failures turn out to be equivalent to the *Byzantine model*, in which arbitrary, correlated and even malicious behavior are all treated as plausible. Computation in this model requires such costly consensus algorithms as to preclude the use of these algorithms in all but the most demanding settings. Moreover, the model requires that all interactions with the outside world be through a Byzantine agreement, which is often impractical. For example, if a system is capable of unlocking a door, the door would have to be controlled at least in quadruplicate, such that three out of four actuators would have to be operated simultaneously to perform the task.[3] Even in extreme settings, such as the control of the space-shuttle cargo hold, triple redundancy was felt to be adequate. Few mundane applications can afford adopt the most pessimistic approach.

To summarize, there seems to be little hope for building practical day-to-day systems capable of tolerating severely incorrect or malicious behavior on the part of some components. Yet, if benign behavior is assumed, one must also consider the possibility that a system will experience failure modes that violate assumptions, and ask what the impact will be and how damage can be minimized.

Design for scale.

Just as it is common to oversimplify issues of fault-tolerance in distributed systems, questions of scale are often neglected. Contemporary distributed systems become hopelessly difficult to manage when more than a few dozen machines are interconnected. Systems that will interconnect hundreds or thousands of machines will require a completely different design mindset, in which scale is viewed as a design feature rather than an aspect that can be dealt with as an afterthought.

Avoid mechanisms that can cascade failures.

[3] In practice, triple modular redundancy is adequate for most applications. Nonetheless, the Byzantine approach requires that there be at least $3T + 1$ total participants in any protocol that will tolerate up to T failures while it is running.

In many current systems, failures can cascade under heavy load or when plausible (but unlikely) failure modes occur. For example, recall the realtime protocols discussed in Chapter 14. In these protocols, a failed component may experience non-atomic broadcast deliveries that corrupt its software state. If such a program were *later* to interact with programs that remained operational, their states could be corrupted too. Many such protocols include lack mechanisms to solve this gradual contamination problem, although most some do provide notification if an obvious error is detected.

A different kind of cascading can occur when machines are declared faulty due to overload. If the operational ones try to take over interrupted tasks, they risk becoming overloaded themselves. This, in turn, would trigger further failures. To avoid such problems one must either design substantial excess capacity into a system (which is often too costly to be practical) or detect overload and react by invoking load-shedding mechanisms. The latter approach is familiar from telephone systems.

Avoid using "magic" mechanisms.

When a large system is built out of large numbers of interacting components, the superficially simple algorithms they embody can misbehave in surprising ways. This poses special problems to the designers of distributed systems, where it is often difficult to predict exactly how a mechanism will behave under real loads. For example, there is a strong temptation to include scheduling heuristics and adaptive mechanisms in low levels of a system; my group did this in some parts the ISIS system for purposes of load balancing. Yet, short of accurately modeling a system, there is no way to know if local optimization decisions will yield globally good behavior, or simply cause the system to "thrash". Given the choice, a simple, well-understood mechanism is always preferable to a fancier but poorly understood one.

20.4. Future directions

The principles enumerated above raise a tremendous number of questions about current and future distributed systems. It is interesting to examine some of the application areas that were covered in the text in this light.

20.4.1. Scaling and administration of file systems.

The major focus of recent work on distributed file systems has been on performance. Systems like Andrew and Sprite represent major advances over, say, the SUN NFS, because they make more effective use of network resources and caching, where effectiveness is typically measured in terms of file transfer bandwidth, access latency, and the number of users the file server can support. These are extremely important issues. But, is it not somewhat narrow to orient file systems

so strongly towards performance considerations?

For example, consider the problem of scaling and administering a large distributed file system. Whereas current file systems use a star architecture, future distributed systems will contain large numbers of file servers of varying capacity, and the performance and capacity of local disks will grow so large that using them just for caching and temporary files will be unacceptably wasteful. Yet, if a file system is assembled out of *multiple* servers, current systems provide little support for management of the ensemble, or for optimizing the assignment of files to available resources. For example, no existing file system maintains the primary copy of a file on the disk local to a user's machine, migrating updates to a remote file server at periods of low load to permit backups from the server and for fault-tolerance. While there has been considerable work on file replication, file systems to date have taken a fairly restricted approach to this whole issue.

This problem is not a purely abstract one. The Cornell Department of Computer Science recently placed an order for 25 workstations which are configured with 350Mbyte local disks. A decision was made to use the local disks only for swapping, temporary files and storage of immutable binaries, because the available file systems otherwise require a great deal of human engineering to manage, and the backup problem would become a major source of overhead. The administrative group was forced to do this because it lacked the personnel to support other general purpose uses of the local disks.

In addition to making more effective use of replication, it is likely that future file systems will need to look hard at semantic information in order to optimize the handling of each file based on its usage patterns. For example, current UNIX-based file systems ignore information about file "type", which forces distributed implementations to guess the best file management policies to use. This policy dates to a period when the UNIX file system was touted for its simplicity. One could question whether simplicity of this sort remains desirable. Most UNIX applications encode information about file type through standard extensions to file names, and the step from this to genuinely typed files is not a huge one. Moreover, information about file type is of great value in a distributed UNIX file system, since it helps in predicting typical modes of access, the likely lifetime of a file, the importance of maintaining availability despite failures, compression methods to use, etc. This list of attributes will surely grow with the widespread use multimedia systems.

Tremendous advantages could be gained by implementing more sophisticated file system architectures. An architecture is needed in which the various servers are knowledgeable about one another and cooperate directly to optimize file distribution in response to patterns of access. Moreover, since this will require some amount of distributed state, the solution must be one which is fault-tolerant and gives well-defined consistency guarantees to file users. Lacking these possibilities, the extent to which file systems can be scaled is inevitably limited.

20.4.2. Security and Authentication in Transactional Contexts

Interesting questions of security and authentication arise in a transactional context.

Consider the authentication issue. In addition to conventional problems of access control and protection, transactional systems depend on the correct use of concurrency control by their components. Moreover, the concurrency control mechanisms must be compatible ones. For example, if a module that uses timestamped concurrency control is called from one that uses locking, applications that include calls to both modules may execute non-serializably. The authentication problem that then arises is to detect concurrency control errors and mismatches in a large system composed of independently developed transactional components.

In current transactional systems, these issues do not arise because components share a common concurrency mechanism. However, in the future, one can easily envision large-scale transactional systems in which no single component provides this function. For example, one may wish to perform transactions under the aegis of CAMELOT on a set of databases managed by multiple commercial database systems. Similarly, one can imagine vendors supplying software packages with transactional interfaces.

Not only is the transactional authentication problem difficult to solve, it is not even clear how one can write down concurrency control requirements or behavior as part of an interface specification. For example, in a module that implements locking and read/write access to a set of variables one might require that a caller acquire a write lock before calling write and a read lock before calling read — *except* when such locks are not needed because some other lock of coarser granularity was previously acquired. How can this even be expressed, much less formally verified?

Next, consider the security problem. Say that a transactional service is accessed by an anonymously implemented caller. Even given a compile-time interface check, one must ask what information can be trusted at runtime. A caller that performs concurrency control incorrectly could contaminate any service that trusts it, and by indirection any other programs that interact with that service. One solution to this problem would employ validated concurrency control and commit "services" accessible over secure RPC. But, one can question whether this is the most efficient and practical solution to the problem. Until system designers begin to ask these sorts of questions and to build systems that include mechanisms such as this, major problems will arise in attempts to move these technologies out of the laboratory.

20.4.3. Replication-Based Systems

Chapter 15 discussed the sorts of group addressing mechanisms and group broadcast mechanisms needed in systems that maintain replicated state. The protocols used for this are complex, and bugs in them could crash large numbers

of machines. It would be hard to envision a failure mode in which a transactional system might shut down every machine on a network, but it is fairly easy to imagine how bugs in a complex protocol could have this effect.

This complexity has several implications. Certainly, work is needed on simplifying the protocols used in replication-based systems. Might there not be a way to build these up from simple, verifiable mechanisms? Another issue is that to the extent possible, these protocols should be implemented as part of the operating system. My reasoning is that if operating systems lack support for the sorts of mechanisms needed in application software, those applications either will not get built, will be built using less than ideal methods or will be forced to individually reproduce the missing mechanisms. Moreover, security and trust considerations would limit the composition of large systems out of smaller components: it will be too hard to agree on protocol specifications and implementation details.

It is clear that a substantial number of applications will need replication. Given this situation, it seems that one would be better off providing such services in a standard way. If individual application builders are asked to take on an effort of this scale, a large amount of duplicated effort will surely result, leading to application software that is less robust, less portable, and harder to maintain than desired.

20.5. Changing the Way People Think about Distributed Computing

Not long ago I met with a vice president of a major corporation who expressed interest in ISIS. This individual explained that within his company, known primarily for its mainframe systems, a perception had arisen that we need to "change the way people think about distributed computing." This comment is intriguing at several levels: how *do* people think about distributed computing? Why change this? And how?

For too long, distributed computer systems have been viewed primarily in terms of interconnection. We have tended to think about such systems as a way to run a program on one machine that uses resources on another and to send mail to our colleagues, and have generally treated workstations as if they were terminals connected to a mainframe. The benefit of a using a workstation is often seen primarily in terms of its ability to offload computation from a centralized resource.

Perhaps the time has come to recognize that centralized servers and transparent distribution are not always good things. It seems clear that the real advantages to distributed systems come about only when one begins to treat the fact of distribution as a positive element that can contribute to the solution of an application, rather than as an annoyance that should be concealed. This does change the requirements that one places on the communication support of the operating system, but the problems that arise can be solved. It is entirely possible that the future successes of distributed computing will be in applications where decentralization, autonomy, fault-tolerance and cooperative behavior are

critical. Whereas many of these applications are today viewed as either too difficult or too costly to solve, the technology for building such systems is finally at hand.

Recall Fredrick Hayes-Roth's comments, quoted in Chapter 14. Like Dorothy who steps out of her aunt's house and suddenly sees the world in color, the revolution in thinking that awaits us is enormous. The potential to solve new problems and take on new applications that this will enable staggers the imagination.

In accepting the challenges offered by these new applications, however, one must be honest concerning just how robust technological solutions to problems can possibly be. And, in situations where analysis of the technical barriers to a solution reveals that the available technology is not adequate to the task, it is necessary to accept the reality of these limitations. Blind faith in technology can simply no longer be justified in the face of the increasingly long list of technical failures that the world has compiled. Those who create these technologies must, for their part, accept the responsibility to do all that they can to ensure that their products will be used in appropriate and reasonable ways.

The era of distributed computing is just beginning. In writing this textbook, a group of us have come together to survey the field and point in some of the directions that it has to offer. In rereading the material that we have produced, it is encouraging to see both by the progress that has been made throughout the field and the accelerating *rate* of progress. At the same time, a tremendous range of problems remain to be solved. In this respect, distributed computing seems to be special within computer science as a whole. Whereas many other areas have flowered rapidly only to stagnate rapidly, new directions keep opening up in distributed systems, and revolutions keep occurring in even the most "traditional" areas of the field. Meanwhile, more and more applications depend on some form of distribution, and these also pose new challenges. The potential applications of distributed computing technology have hardly been tapped. Provided that this is done in a careful, considered manner, distributed computing could change the way that we deal with computer systems in a profound and beneficial way.

It is unfortunate that the potential for abuse of this technology is a real as for any other technology that civilization has devised. Nonetheless, that potential *is* real, and will have increasingly important consequences. Our field must accept the responsibility for this technology: even as we create these new forms of computing, it falls upon us to control them through new and more demanding ethical standards.

20.6. Acknowledgments

I am grateful to Robert Cooper, Ajei Gopal and Barbara Simons for commenting on drafts of this material. I also feel that an apology is due to the authors of papers and books germaine to this discussion. I could certainly have cited relevant technical material, but felt that this would be inappropriate given the

theme of the chapter. And, I know of little material on ethics in relation to technologies of this sort. I would be grateful for any references that readers knowledgeable about the subject might care to recommend.