Engineering Fault-Tolerant Distributed Computing Systems†

Özalp Babaoğlu

86-755

May 1986
(Revised June 1987)

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

†Partial support for this work was provided by the National Science Foundation under Grant DCR-86-01864 and AT&T under a Foundation Grant.

Engineering Fault-Tolerant Distributed Computing Systems*

Özalp Babaoğlu

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

ABSTRACT

We view the design of fault-tolerant computing systems as an engineering endeavor. As such, this activity requires understanding the theoretical limitations and the scope of the feasible designs. We survey the impact that various environment characteristics and design choices have on the resultant system properties. We propose a single metric—the system reliability—as an appropriate measure for exploring tradeoffs among a potentially-large design space.

June 25, 1987

* Partial support for this work was provided by the National Science Foundation under Grant DCR-86-01864 and AT&T under a Foundation Grant.

1. Introduction

Continued and correct operation in the presence of failures are required attributes for an increasing number of computing systems [Kim84, Spec84]. Unfortunately, given a finite amount of hardware, it is impossible to construct a computing system that never fails. The best we can hope to achieve are systems that continue correct operation "with high probability."

There are two complementary strategies for coping with failures. The first is to construct a computing system from components that are less likely to fail—the *fault-avoidance* approach. The second is to construct a system that continues to function correctly despite failures—the *fault-tolerance* approach [RLT78]. The fault-avoidance approach by itself is limited only by technological and economic factors, and presents no real conceptual challenges at the system design level. On the other hand, the fault-tolerance approach requires techniques for transparently masking failures or restarting the computation from some past state. In this paper, we consider only fault tolerance through replication in computing systems that are viewed at a level where the "components" consist of processors (each with associated memory and input/output devices) and communication hardware. We do not consider failures due to faulty software.

Informally, a *distributed computing system* is a collection of autonomous processors that share no memory, do not have access to a global clock and communicate only by exchanging messages. Recent developments in hardware technology, along with the inherent distributed nature of many applications have made distributed systems a cost effective alternative to centralized computing systems. Distributed algorithms rely on cooperation among processors. The lack of shared memory and random communication delays contribute to the difficulty of programming distributed systems. When processors are allowed to fail, this task becomes even more difficult.

Recently, considerable effort has been devoted to identifying the appropriate primitives and structures towards a "methodology" of fault-tolerant computing [Lamp84, SL85]. We now have the understanding necessary to incorporate fault tolerance into a large class of applications without having to reinvent algorithms for each special case. Our goal is to extend this methodology for designing fault-tolerant distributed computing systems so that it resembles the traditional "engineering" endeavor. This entails solving the appropriate sub-problems, understanding the interactions between the proposed solutions and the properties of the resulting system, and developing the measures necessary for exploring as wide a range of designs as possible in searching trade-offs.

We make the analogy to engineering an aircraft. This activity is based on theoretical foundations such as physics, aerodynamics, fluid mechanics, etc. It requires an understanding of the interactions between the numerous design parameters (e.g., construction material, wing span, number of engines, etc.) and the properties of the resultant design (e.g., maximum speed, pay load capacity, maximum altitude, etc.). Whether the final design resembles a hang glider or a wide-body Airbus is simply due to the tradeoffs that are made among the many possible designs that satisfy the requirements. In engineering fault-tolerant computing systems, we have the necessary theoretical foundations. What we lack is an understanding of the interactions between the design parameters and the system properties. This is essential if we hope to effectively explore the design space in search of tradeoffs.

Typically, the properties of a fault-tolerant system design are given in terms of the maximum number of processors that can fail (resiliency), the total number of messages that are exchanged (communication complexity), the number of rounds of message exchanges (time complexity) and the amount of computation performed during each round (computational complexity). Unfortunately, it is rarely the case that one design dominates another with respect to all four of these measures. In most cases, one cost can be traded for another

cost. Consequently, it is quite difficult to say when one design is "better" than another design. Ultimately, the design goal for a fault-tolerant system is to guarantee a lower bound on the probability that it will perform the right action after it has been operating for some length of time. This metric, which we call the system *reliability*, subsumes all four cost measures defined above such that it is suitable for exploring tradeoffs. Certainly, for many applications, other requirements such as total cost and performance may be equally important. For simplicity, we only consider the system reliability in evaluating alternative solutions.

In the next section we outline the replicated system structure that constitutes the core of the fault-tolerant computing system design methodology. In comparing alternative solutions for the same problem, we distinguish between effects that are due to the characteristics of the external environment (and, thus, are usually beyond the control of the designer) and those that are due to choices made for the design parameters. In Section 3, we survey the impact that the external environment characteristics have on fault-tolerant systems. Then, in Section 4, we examine the alternative solutions that are possible by varying the design parameters. The reliability metric we propose reveals some subtle and counterintuitive interactions between certain system design parameters and the resulting fault tolerance.

2. System Structure

Our goal is to reliably execute a single program that performs some arbitrary computation. Without loss of generality, we assume that the application cyclically reads data from an input source, performs a deterministic computation based on this data, and writes outputs. This structure for a computation has been called a *state machine* [Lamp78, Schn87].

To achieve fault tolerance, the state machine (including the input sources) is replicated and each instance executes on its own processor [Lamp84, SL85, GPD84]. Fault tolerance requirements usually dictate that the replicated system not rely on the correctness of any single component for its correct operation. Furthermore, the independence of processor failures necessitates that the processors be isolated, both physically and electrically. Consequently, when viewed at an appropriate level of abstraction, the replicated system has the same properties as a distributed system—autonomous processors with no shared resources that communicate through a network.

In the rest of the paper, we refer to a processor executing a state machine instance simply as a *processor*. The entire ensemble is called the *system*. An application-dependent threshold Ψ dictates the minimum number of correct processors that must exist to maintain correctness in the presence of failures.

If the same result is to be generated by all correct processors, they must use the same input value for their computation. This seemingly simple requirement is not always easy to achieve. For example, at any given time, correct analog sensors may differ slightly in their readings. In addition, processors may read their associated sensors at slightly different times. Therefore, since the processors cannot simply compute based on local data, they must execute a protocol to exchange local inputs and to decide on a common value to use as *the* input for the computation. In the presence of processor failures, a protocol that achieves this goal is called a *distributed consensus protocol* [PSL80, Fisc83, SD83, articles in this book]. Formally, in a system with n processors where each processor i has an initial local value v_i, a protocol achieves consensus if upon termination each processor computes an n-element vector such that the following two conditions are satisfied:

C1: (Agreement) All correct processors compute a common vector $\mathbf{W}=(w_1, w_2, \cdots, w_n)$;

C2: (Validity) The jth component of the vector is equal to the initial local value of processor j if j is correct (i.e., $w_j = v_j$ for each j corresponding to a correct processor).

Typically, the common input value to be used for the computation is obtained by each processor's applying the same deterministic function (such as arithmetic mean or median) to the vector **W**. This protocol for input dissemination can be easily extended to ensure that all elements of the consensus vector correspond to input values for the same iteration of the state machines [Schn82].

The correctness specification for each state machine replica is given as an input-output relation defined over all possible input vectors **W**. We assume that each processor is correctly programmed in the sense that it implements this relation in the absence of failures. We say that a *system is correct* if at least Ψ processors generate an output for the computation that is consistent with the input-output specification. The *reliability* of a system at time T is the probability that it is correct at time T given that it was initially correct.

Each of the n processors in the system can be in two states: correct and faulty. Each processor starts out in the *correct state* in which its behavior conforms to the specification encoded as a program. After some time, a processor may *fail*; thereafter it is called *faulty*. Note that "correctness" and "faultiness" are only classifications of the internal state of a processor. A *faulty* processor may deviate from its specification. We classify failures according to the deviations that are permitted.

We assume that the times processors spend in the correct state before becoming faulty are independent and identically distributed exponential random variables with rate λ. In other words, each processor fails independently of all the others at a common constant failure rate. The exponential distribution assumption for failure times has been empirically justified for a large class of components, including electronic hardware, that do not "age" [SS82]. The independence of failures is typically achieved through physical and electrical isolation of the processors. Without loss of generality, we assume that $\lambda = 1$ in the rest of the paper by scaling all time variables in our discussion by $1/\lambda$. Since $1/\lambda$ is the expected value of an exponential random variable with rate λ, the unit of time in our results is the mean-time-to-failure interval of a single processor. We do not consider the possibility of repairing faulty processors.

3. Environment Characteristics

The global system structure implied by the methodology presented in the previous section is as follows. An application which is not tolerant to processor failures is replicated on each of the n processors. Each processor reads a value from its local input source. The ensemble executes a consensus protocol to exchange input values and to agree on the unique value to be used for the computation. Each processor independently computes the output corresponding to the input value. If there are sufficiently many processors generating correct outputs, the system state is correct.

While the computation itself is performed by each processor in isolation, the consensus protocol requires cooperation among the processors and is highly sensitive to the properties of the environment in which it is formulated. We survey these properties below.

3.1. Synchrony

We say that a system is *synchronous* if the relative processor speeds and the network message delivery delays are bounded and these bounds are known by the processors. A system is called *completely asynchronous* if neither the relative processor speeds nor the message delivery delays are bounded. A *deterministic* protocol is one that takes only

deterministic steps in its computation. Alternatively, a *randomized* protocol can take computational steps based on non-deterministic events (such as coin tosses). It is known that the consensus problem has no deterministic solution in completely asynchronous systems in the presence of even a single failure [FLP85] †. Fortunately, most realistic systems do exhibit bounded relative processor speeds and message delivery delays. We consider only synchronous systems in the rest of our discussions.

3.2. Faulty Processor Behavior

Before we can proceed with the design of a fault-tolerant system, we must decide on an adequate characterization of the behavior of faulty processors. We describe such behavior as deviations from the protocol that they are supposed to be executing. Based on this classification, we distinguish three failure models:

1. *Crash Failure:* A processor stops executing its protocol, never to resume again. Note that, since the only externally-visible behavior of processors in a distributed system is the messages they send, a crash failure is equivalent to a processor stopping to send any more messages. Internally, it could be doing any arbitrary computation (e.g., executing an infinite loop).
2. *Omission Failure:* A processor fails to send some of the messages it is supposed to send. The messages it does send are always correct.
3. *Byzantine (Malicious) Failure:* A faulty processor exhibits arbitrary behavior. In particular, it may send messages not prescribed by its protocol, fail to send others, collude with other faulty processors to confound the system and behave in any other arbitrary manner.

While crash failures are more restrictive than omission failures, the two failure models are equivalent with respect to the consensus problem in the sense that a solution for one can be transformed into a solution for the other [Hadz84]. For both failure models, consensus can be achieved in the presence of any number of faulty processors. The lower and upper bounds for the time complexity of consensus under these fault models coincide at $t+1$ rounds, where t is the protocol resiliency [Fisc83, Hadz84].

Byzantine failures represent a "non-assumption" about the behavior of faulty processors. Achieving consensus in the presence of Byzantine processor failures is considerably more expensive than with the other two failure models. In general, at least $3t+1$ processors are required to tolerate up to t faulty ones [SPL80]. Any protocol must execute for at least $t+1$ rounds to reach consensus [FL82]. While $t+1$ rounds is a lower bound, currently no consensus protocol for Byzantine failures achieves this time complexity while exchanging only a polynomial number of bits.

The reliability of the system when subjected to crash or omission failures is simply the probability that at least one out of the n processors generates an output at the end of the computation. In other words, we have the correctness threshold $\Psi=1$. Therefore, the reliability of a replicated system for these failure models will always be greater than that of a single processor, independent of the level of replication. On the other hand, the system reliability in the presence of Byzantine processor failures is given as the probability that the consensus protocol succeeds and that at least Ψ of the processors remain correct throughout the computation phase. We will evaluate these expressions in Section 4. Note that if the desired response from the system is a single output to the external world despite up to t Byzantine failures, the replicated system cannot have a correctness threshold less

† Recently, Dolev *et al.* have given a much finer characterization of the minimal synchrony necessary for the solvability of the consensus problem [DDS87].

than majority (i.e., $\Psi > t$) and it must rely on a single voter to implement it [GPD84].

3.3. Authentication

The results we presented above are relevant for systems in which a faulty processor can both undetectably forge messages on behalf of other processors and also tamper with the contents of messages it is forwarding. If the system supports an authentication mechanism (such as digital signatures [DH76]), then the undetectable behavior of even Byzantine processors is severly restricted. In fact, in a system with authentication, a consensus protocol can tolerate any number of Byzantine processor failures [LSP82]. Furthermore, consensus can be reached in $t+1$ rounds, thus matching the lower bound for time complexity [DS83].

Let us examine the reliability of the state machine ensemble with authentication and Byzantine processor failures. While the consensus protocol for input dissemination can tolerate any number of failures, the correctness of the system can be guaranteed only if the number of faulty processors by the end of the computation phase represents a minority of the total number of processors. Unlike crash or omission failures, a processor exhibiting Byzantine failure can still lie about its own state even with authentication.

3.4. Communication Model

Up to this point, we have assumed that the network used for communication among processors is a perfectly-reliable, fully-connected point-to-point graph. Even if the network remains perfectly reliable, we cannot achieve consensus in systems that communicate through sparsely-connected networks. It is known that the connectivity of the network must be at least $2t+1$ without authentication and $t+1$ with authentication [Dole82].

One simple technique to deal with unreliable network components is to model the failure of a link between two processors as the failure of one of the processors. While not requiring any new mechanisms, this technique results in overly-pessimistic designs, since t is now the sum of the number of processor and link failures. More realistic designs can be obtained by introducing new failure models that explicitly deal with communication failures [PT86].

Recently, we have studied the time complexity of consensus protocols in systems that communicate through networks other than point-to-point graphs [BSD87]. A typical architecture for distributed systems consists of several processor clusters on a common network in which the intra-cluster communication takes place over a shared, multiple-access media that support broadcasts [Stal84]. This broadcast network-based architecture encompasses a wide range of designs. For example, the clusters could represent geographically distant local area networks connected through gateways (such as the Xerox Internet comprising a large number of Ethernets [MB76]). Alternatively, each cluster could be a single, tightly-coupled multiprocessor with an internal bus interconnect and inter-cluster links implemented as bus adaptors.

Regardless of the physical realization of the architecture, we can abstract the behavior of communication in such systems as follows:

BNP: *(Broadcast Network Property)* In response to a broadcast, all processors that receive a message receive the same message.

This property ensures that for all possible failures, a processor cannot send conflicting messages to other processors in a single broadcast. For a given broadcast network, the set of processors that receive the (same) message in response to a broadcast is called the *receiving set*. The *broadcast degree* of a network is defined to be a lower bound on the size of the receiving sets for all broadcasts in the network. The receiving sets may vary from one

broadcast message to another as well as from one sender to another. For a network to have broadcast degree b, all we require is that each of these sets contain at least b processors. We assume that every processor receives its own broadcasts regardless of failures. Note that communication failures manifest themselves in defining a particular broadcast degree for the network.

Given this framework, we have shown that in a system with n processors where the broadcast degree is b, consensus can be achieved in 2 rounds in the presence of Byzantine failures and no authentication as long as $b > t + n/2$ [BD85]. For broadcast degrees smaller than t, achieving consensus in the presence of Byzantine failures requires $t+1$ rounds, just as it does in point-to-point networks [BDS87]. For omission failures, the condition $b \geq t$ suffices to achieve a 2-round solution. Furthermore, in the case of omission failures, we were able to obtain a parametrized solution that achieves consensus in $t - b + 3$ rounds for any broadcast degree $2 \leq b < t$ [BDS87]. The networks that are characterized by this range of broadcast degrees span the entire spectrum from point-to-point graphs to full broadcast networks. These results reveal a new dimension in the design space of fault-tolerant systems—performance and resiliency can be traded for network cost and complexity.

4. Design Parameters

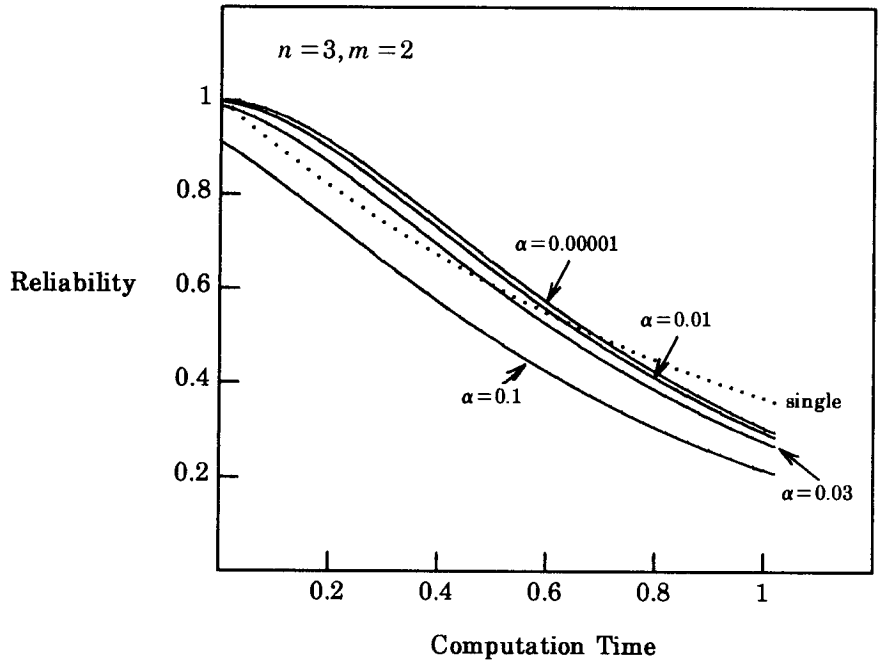
Having defined the context in which a fault-tolerant system is to be designed, we next examine the properties of the resultant system as a function of the choices that are made for the various design parameters.

4.1. Replication Level

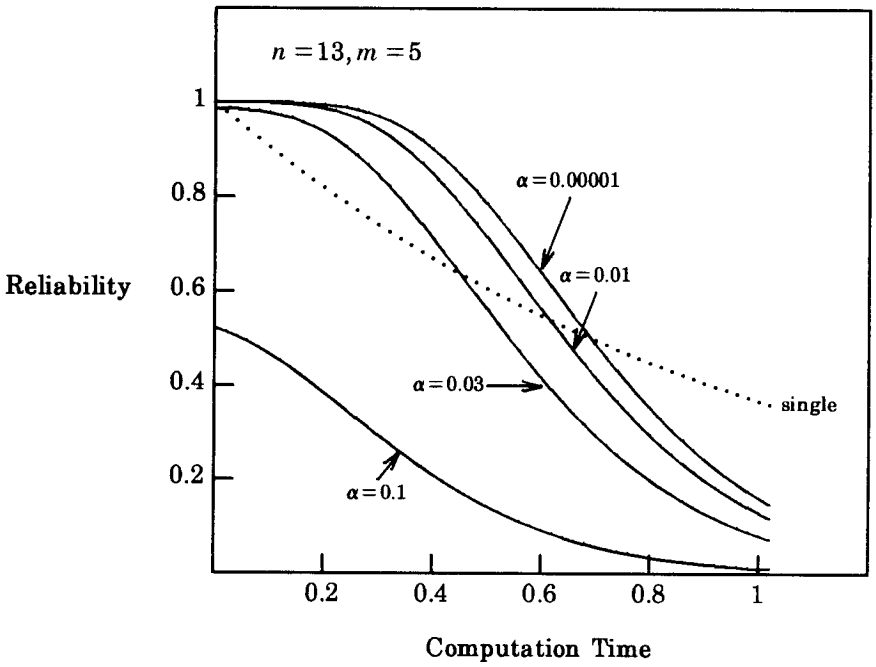
Perhaps the first question that the designer of a fault-tolerant computing system must answer is: How many times should the state machine be replicated? Although this question has economic as well as reliability implications, we will address only the reliability issues. Recall that if the design specification of the system is given in terms of the resiliency (t), then the characteristics of the external environment dictate the minimum level of replication. For example, with Byzantine failures and no authentication, $n > 3t$. If the system supports authentication, then $n > 2t$. If the failures are restricted to crash or omission, then the replication level need only be one greater than the resiliency ($n > t$).

If the design specification of the system is given in terms of reliability, the answer to the replication level question is not so clear. It is well known that replicated systems can have worse reliability than their non-replicated counterparts for certain system parameters [SS82]. Intuitively, as there are more processors in the system due to replication, there are more sources of failures. Consequently, the probability that enough processors will be correct arbitrarily late in the computation phase will be smaller than the probability of a single processor's being correct.

The reliability of state machine ensembles indeed exhibit this behavior as a function of the replication level. Let α denote the time required for the system to implement a round. In Figure 1, we plot the reliability of systems subject to Byzantine failures with authentication and with various levels of replication and round lengths where the correctness threshold, Ψ, is majority. In the figures, m denotes the number of rounds that the consensus protocol is executed to disseminate the input. For the derivations of the expressions used to generate the figures in this paper, we refer the reader to [Baba87b]. Recall that the unit of time in our results is the mean-time-to-failure of a single processor. With today's technology, it is trivial to achieve mean-time-to-failures for processors in excess of 100 hours. Typically, processor and communication network speeds allow rounds to be realized in at most several seconds. Consequently, realistic normalized (i.e., after scaling by the mean-time-to-failure) values for α are of the order 10^{-5}. The figures include larger values



(a)



(b)

Fig. 1. Reliability of state machine ensembles.

for the round length simply to explore as large a design space as possible.

The reliability of the non-replicated design alternative is labeled "single" in the figures. We note that for different parameter values, there are distinct intervals for the computation time in which the non-replicated single processor has higher reliability than the replicated system. In fact, for sufficiently long computation times (perhaps unrealistically long), the single processor has uniformly higher reliability than the replicated system. As the replication level is increased, the computation time that delineates highly reliable systems from highly unreliable systems becomes more sharply defined. We can formally show that as $n \to \infty$ and $\alpha \to 0$, the system remains perfectly reliable during computation times up to $\ln(n/(\Psi-1))$ and is incorrect with certainty for computations that last longer than this time [Baba87b]. In case of majority threshold, this transition time has the limit $\ln(2)=0.693$.

4.2. Consensus Protocol Running Time

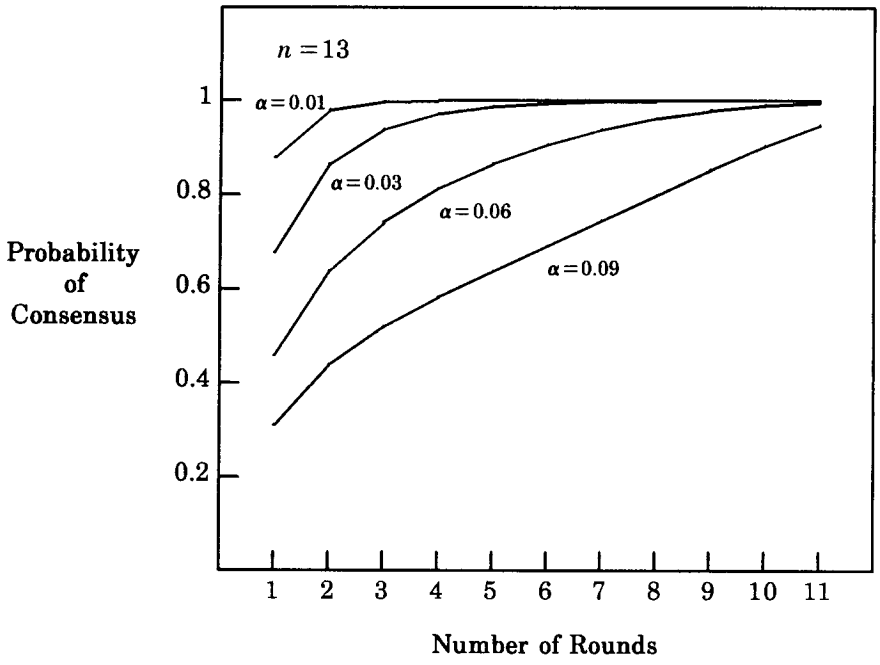
In a system with n processors and Byzantine failures, an m-round consensus protocol execution is at least $m-1$ resilient in the sense that consensus is guaranteed provided there are no more than $m-1$ faulty processors (i.e., $t<m$). Given our model where processors fail independently according to an exponential distribution, there could be m-round executions where consensus is achieved despite the fact that protocol resiliency is violated (i.e., $t \geq m$). In [Baba87a] we characterize such executions and derive expressions for the probability of their occurrence. Using these results, in this section we will explore the tradeoffs that are involved in selecting the running time of the input consensus protocol.

All other factors being equal, an $(m+1)$-round execution has a higher probability of achieving consensus than an m-round execution. Figure 2a displays the probability with which an m-round execution achieves consensus in a system with 13 processors and various round lengths. Note that for small (realistic) values of α, a 2 to 3-round execution defines the point diminishing returns—the system achieves consensus with high probability for this execution length and any additional rounds of execution result in negligible gain. However, for large values of α, each additional round of execution increases the probability of consensus by a significant amount.

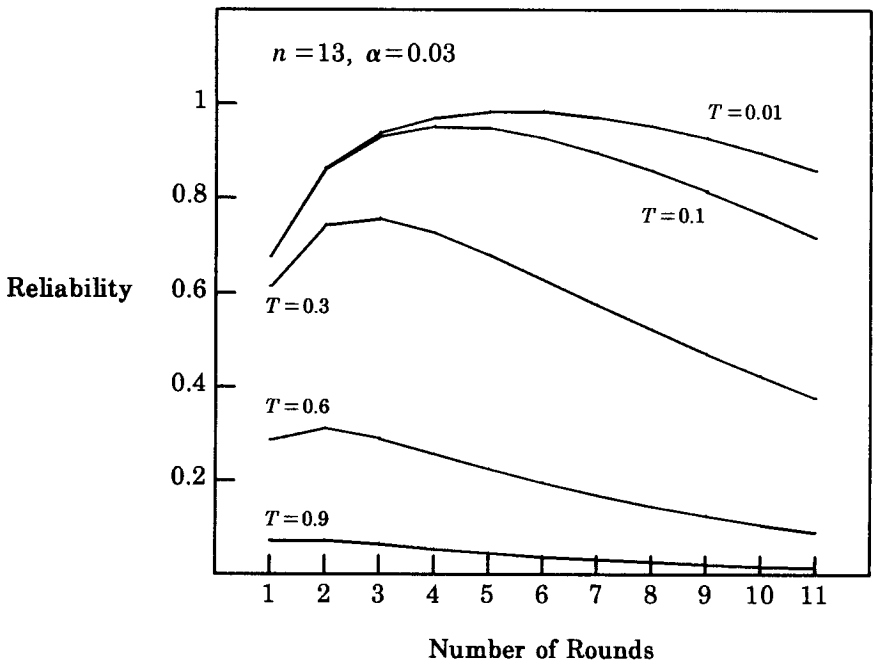
Next we investigate the effect of input consensus protocol execution time on the reliability of a state machine ensemble. Although the consensus protocol itself always benefits from additional rounds of execution, this benefit is gained at the expense of having fewer correct processors available for the application computation. Thus, the increase in the probability of the consensus phase succeeding may be offset by the decrease in the probability of the system achieving the correctness threshold. Figure 2b is the reliability of a system with 13 processors as a function of the input consensus protocol running time and various computation times. When the computation time is short with respect to the consensus phase, the system reliability increases for up to about 5 rounds of consensus execution and then starts to decrease. For computation times that are longer, the consensus protocol execution length that maximizes system reliability becomes shorter. In fact, for sufficiently long computations, the reliability is a monotone decreasing function of m, suggesting that the system is better off not wasting its time with consensus and commencing the computation immediately after a single round of input data exchange.

4.3. Fault Detection

In Section 4.2 we saw that in a system with Byzantine failures, a state machine ensemble became arbitrarily unreliable after sufficiently long computation times. This is due to the accumulation of faulty processors in the system such that there comes a time when the failure of a single additional processor tips the scale away from the correctness



(a)



(b)

Fig. 2. Effect of consensus protocol running time.

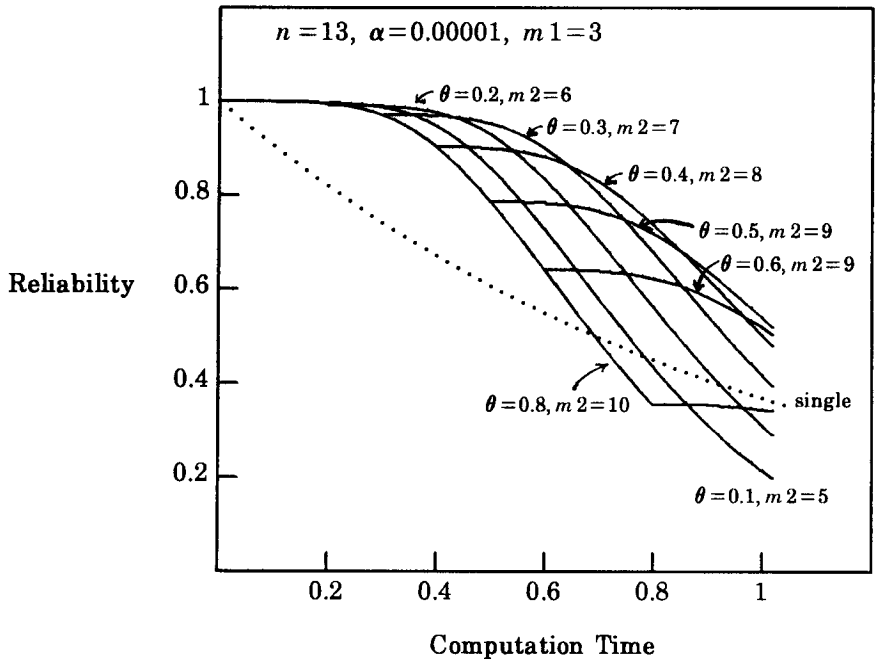


Fig. 3. The effect of fault detection.

threshold. Intuitively, we need to be able to periodically identify the processors that have failed and remove them from the system.

In this section, we study system reliability in the presence of a protocol that can detect all processors that have failed by θ time units into the application computation and remove them from the system. In [Baba87b] we discuss methods that are suitable for implementing such a fault detector based on consensus protocols. In Figure 3 we illustrate the effect of executing a fault detection protocol θ units into the application computation for a system with 13 processors. The consensus protocol used for input dissemination is always executed for $m1$ rounds ($m1=3$ in the example) while that used for fault detection is executed a variable number of rounds (denoted $m2$) depending on the value of θ. Note that the system in Figure 3 has reliabilities similar to those depicted in Figure 1b until time $T=\theta$. At this point, the system maintains its current reliability level for some time and then resumes deterioration. If fault detection is performed early enough in the computation, high system reliability can be maintain much longer than is possible without fault detection. Obviously, the technique could be extended to perform fault detection every θ units of the computation rather than only once.

5. Conclusions

We have presented the reliability of a fault-tolerant computing system as a metric that is more suitable than resiliency in evaluating different designs. Not only is the reliability metric more in tune with the design specifications for most applications, it also captures most of the subtle and counterintuitive interactions between design parameters and system properties.

We surveyed the implications of external environment characteristics such as failure models, synchrony, authentication and communication models on fault-tolerant system

properties. We then examined the properties of alternative solutions due to design decisions. We showed that increases in the replication level serve to more sharply delineate computation times that separate highly reliable systems from highly unreliable ones. One of our counterintuitive results involves selecting the running time of the consensus protocol that is used to disseminate the inputs to the state machine replicas. For certain system parameters, we have seen that increasing the protocol running time actually decreases the overall system reliability. Finally, we have shown that the ability to detect faulty processors and remove them from the system is extremely effective with respect to reliability.

Our results are a first step towards completing the design methodology for building fault-tolerant computing systems. Although we were able to evaluate several design choices individually using the system reliability as the criterion, a realistic design endeavor has to cope with alternatives resulting from varying several parameters simultaneously. Furthermore, the evaluation criterion has to include measures other than reliability such as performance. We feel that our approach is suitable for extension in any one of these directions.

Acknowledgments

I am grateful to Fred Schneider, Rogério Drummond and Pat Stephenson for discussions that helped me formulate many of the ideas surveyed in this paper. Comments from Barbara Simons helped improve the presentation.

References

Baba87a Ö. Babaoğlu, Stopping times of distributed consensus protocols: a probabilistic analysis. *Information Processing Letters*, vol. 25, no. 3, pp. 163-169, (May 1987).

Baba87b Ö. Babaoğlu, On the reliability of consensus-based fault-tolerant distributed computing systems. *ACM Trans. on Computer Systems*, (to appear).

BD85 Ö. Babaoğlu and R. Drummond, Streets of Byzantium: Network architectures for fast reliable broadcasts. *IEEE Trans. Software Eng.*, vol. SE-11, no. 6, pp. 546-554, June 1985.

BDS87 Ö. Babaoğlu, R. Drummond and P. Stephenson, Reliable broadcast protocols and communication models: Tradeoffs and lower bounds. *Springer-Verlag Distributed Computing*, (to appear).

DH76 W. Diffie and M. Hellman, New directions in cryptography. *IEEE Trans. on Inf. Theory*, vol. IT-22, pp. 644-654, 1976.

Dole82 D. Dolev, The Byzantine Generals strike again. *Journal of Algorithms*, vol. 3, no. 1, pp. 14-30, 1982.

DDS87 D. Dolev, C. Dwork and L. Stockmeyer, On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, vol. 34, no. 1, pp. 77-97, January 1987.

DS83 D. Dolev and H. R. Strong, Authenticated algorithms for Byzantine Agreement.

SIAM J. Comput., vol. 12, no. 4, pp. 656-666, November 1983.

Fisc83 M. J. Fischer, The consensus problem in unreliable distributed systems (A Brief Survey). Tech. Rep. YALEU-DCS-RR-273, Dept. of Computer Science, Yale University, New Haven, Connecticut, June 1983.

FL82 Fischer, M. and Lynch, N. A lower bound for the time to assure interactive consistency. *Inform. Proc. Letters 14*, no. 4, pp. 183-186, April 1982.

FLP85 M.J. Fischer, N.A. Lynch and M.S. Paterson, Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, vol. 32, no. 2, pp. 374-382, April 1985.

GPD84 H. Garcia-Molina, F. Pittelli and S. Davidson, Applications of Byzantine Agreement in database systems. Tech. Rep. TR 316, Princeton University, Princeton, New Jersey, June 1984.

Hadz84 V. Hadzilacos, Issues of fault tolerance in concurrent computations. Ph.D Thesis, Tech. Rep. TR-11-84, Aiken Computation Laboratory, Harvard University, Cambridge, Mass., June, 1984.

Kim84 W. Kim, Highly available systems for database applications. *ACM Computing Surveys*, vol. 16, no. 1, pp. 71-98, March 1984.

Lamp84 L. Lamport, Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. on Programming Languages and Systems*, vol. 6, no. 2, pp. 254-280, April 1984.

MB76 R. Metcalfe and D.R. Boggs, Ethernet: Distributed packet switching for local computer networks. *Commun. ACM*, vol. 19, no. 7, pp. 396-403, July 1976.

PSL80 M. Pease, R. Shostak and L. Lamport, Reaching agreement in the presence of faults. *Journal of the ACM*, vol. 27, no. 2, pp. 228-234, April 1980.

PT86 K.J. Perry and S. Toueg, Distributed agreement in the presence of processor and communication faults. *IEEE Trans. on Software Engineering*, vol. SE-12, no. 3, pp. 477-482, March 1986.

RLT78 B. Randell, P.A. Lee, and P.C. Treleaven, Reliability issues in computing system design. *ACM Computing Surveys*, vol. 10, no. 2, pp. 123-166, June 1978.

Schn82 F. B. Schneider, Synchronization in distributed programs. *ACM Trans. Programming Languages and Systems*, vol. 4, pp. 125-148, April 1982.

Schn87 F. B. Schneider, The state machine approach: A tutorial. This volume.

SL85 F. B. Schneider and L. Lamport, Paradigms for distributed programs. In *Distributed Systems: Methods and Tools for Specification*, Paul, M. and Siegert H.J. (Eds.), Springer-Verlag Lecture Notes in Computer Science Vol. 190.

SS82 D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*. Digital Press, Belford, Mass. (1982).

Spec84 A. Z. Spector, Computer software for process control. *Scientific American*, vol. 251, no. 3, pp. 174-187, September 1984.

Stal84 Stallings, W. Local networks. *ACM Computing Surveys*, vol. 16, no. 1, pp. 3-41, March 1984.

SD83 H.R. Strong and D. Dolev, Byzantine agreement. In *Digest of Papers, Spring Compcon 83*, San Francisco, California, pp. 77-81, March 1983.