Chapter 2:

What Good are Models

and

What Models are Good?

Fred B. Schneider*

Department of Computer Science
Cornell University
Ithaca, New York 14853 U.S.A.

1. Refining Intuition

Distributed systems are hard to design and understand because we lack intuition for them. Perhaps this is because our lives are fundamentally sequential and centralized. Perhaps not. In any event, distributed systems are being built. We must develop an intuition, so that we can design distributed systems that perform as we intend and so that we can understand existing distributed systems well enough for modification as needs change.

Although developing an intuition for a new domain is difficult, it is a problem that engineers and scientists have successfully dealt with before. We have acquired intuition about flight, about nuclear physics, and about chaotic phenomena (like turbulence). Two approaches are conventionally employed:

> **Experimental Observation**. We build things and observe how they behave in various settings. A body of experience accumulates about approaches that work. Even though we might not understand why something works, this body of experience enables us to build things for settings similar to those that have been studied.
>
> **Modeling and Analysis**. We formulate a model by simplifying the object of study and postulating a set of rules to define its behavior. We then analyize the model—using mathematics or logic—and infer consequences. If the model accurately characterizes reality, then it becomes a powerful mental tool.

Both of these approaches are illustrated in this text. Some chapters report experimental observation; others are more concerned with describing and analyzing models. Taken together, however, the chapters constitute a collective intuition for the design and analysis of distributed systems.

---

*This material is based on work supported in part by the Office of Naval Research under contract N00014-91-J-1219, the National Science Foundation under Grant No. CCR-8701103, and DARPA/NSF Grant No. CCR-9014363. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

In a young discipline—like distributed systems—there is an inevitable tension between advocates for "experimental observation" and those for "modeling and analysis". This tension masquerades as a dichotomy between "theory" and "practice". Each side believes that theirs is the more effective way to refine intuition. Practitioners complain that they learn little from the theory. Theoreticians complain that practitioners are not addressing the right problems. A theoretician might simplify too much when defining a model; the analysis of such models will rarely enhance our intuition. A practitioner might incorrectly generalize from experience or concentrate on the wrong attributes of an object; our intuition does not profit from this, either. On the other hand, without experimental observation, we have no basis for trusting our models. And, without models, we have no hope of mastering the complexity that underlies distributed systems.

The remainder of this chapter is about models for distributed systems. We start by discussing useful properties of models. We then illustrate that simple distributed systems can be difficult to understand—our intuition is not well developed—and how models help. We next turn to two key attributes of a distributed system and discuss models for these attributes. The first concerns assumptions about process execution speeds and message delivery delays. The implications of such assumptions are pursued in greater detail in Chapter 4 (???Ozalp+Marzullo) and Chapter 5 (???Ozalp+Sam). The second attribute concerns failure modes. This material is fundamental for later chapters on implementing fault tolerance, Chapter 6 (???FS state machine) and Chapter 7 (???FS et al primary-backup). Finally, we argue that all of these models are useful and interesting, both to practitioners and theoreticians.

2. Good Models

For our purposes, a *model* for an object is a collection of attributes and a set of rules that govern how these attributes interact. There is no single correct model for an object. Answering different types of questions about an object usually requires that we employ different models, each with different attributes and/or rules. A model is *accurate* to the extent that analyzing it yields truths about the object of interest. A model is *tractable* if such an analysis is actually possible.

Defining an accurate model is not difficult; defining an accurate and tractable model is. An accurate and tractable model will include exactly those attributes that affect the phenomena of interest. Selecting these attributes requires taste and insight. Level of detail is a key issue.

In a tractable model, rules governing interactions of attributes will be in a form that supports analysis. For example, mathematical and logical formulas can be analyzed by uninterpreted symbolic manipulations. This makes these formalisms well suited for defining models. Computer simulations can also define models. While not as easy to analyze, a computer simulation is usually easier to analyize than the system it simulates.

In building models for distributed systems, we typically seek answers to two fundamental questions:

Feasibility. What classes of problems can be solved?

Cost. For those classes that can be solved, how expensive must the solution be?

Both questions have practical import as well as having theoretical value. First, being able to recognize an unsolvable problem lurking beneath a system's requirements can head-off wasted effort in design, implementation, and testing. Second, knowing the cost implications of solving a particular problem allows us to avoid designs requiring protocols that are inherently slow or expensive. Finally, knowing the inherent cost of solving a particular problem provides a yardstick with which we can evaluate any solution that we devise.

By studying algorithms and computational complexity, most undergraduates learn about undecidable problems and about complexity classes, the two issues raised above. The study builds intuition for a particular model of computation—one that involves a single sequential process and a uniform access-time memory. Unfortunately, this is neither an accurate nor useful model for the systems that concern us. Distributed systems comprise multiple processes communicating over narrow-bandwidth, high-latency channels, with some processes and/or channels faulty. The additional processes provide more computational power but require coordination. The channel bandwidth limitations help isolate the effects of failures but also mean that interprocess communication is a scarce system resource. In short, distributed systems raise new concerns and understanding these requires new models.

A Coordination Problem

In implementing distributed systems, process coordination and coping with failures are pervasive concerns. Here is an example of such a problem.

> **Coordination Problem**. Two processes, *A* and *B*, communicate by sending and receiving messages on a bidirectional channel. Neither process can fail. However, the channel can experience transient failures, resulting in the loss of a subset of the messages that have been sent. Devise a protocol where either of two actions α and β are possible, but (i) both processes take the same action and (ii) neither takes both actions. ■

That this problem has no solution usually comes as a surprise. Here is the proof.

> Any protocol that solves this problem is equivalent to one in which there are rounds of message exchange: first *A* (say) sends to *B*, next *B* sends to *A*, then *A* sends to *B*, and so on. We show that in assuming the existence of a protocol to solve the problem, we are able to derive a contradiction. This establishes that no such protocol exists.
>
> Select the protocol that solves the problem using the fewest rounds. By assumption, such a protocol must exist, and, by construction, no protocol solving the problem using fewer rounds exists. Without loss of generality, suppose that *m*, the last message sent by either process, is sent by *A*.
>
> Observe that the action ultimately taken by *A* cannot depend on whether *m* is received by *B*, because its receipt could never be learned by *A* (since this is the last message). Thus, *A*'s choice of action α or β does not depend on *m*. Next, observe that the action ultimately taken by *B* cannot depend on whether *m* is received by *B*, because *B* must make the same choice of action α or β even if *m* is lost (due to a channel failure).
>
> Having argued that the action chosen *A* and *B* does not depend on *m*, we conclude that *m* is superfluous. Thus, we can construct a new protocol in which one fewer message is sent. However, the existence of such a shorter protocol contradicts the assumption that the protocol we employed used the fewest number of rounds. ■

We established that the Coordination Problem could not be solved by building a simple, informal model. Two insights were used in this model:

(1) All protocols between two processes are equivalent to a series of message exchanges.

(2) Actions taken by a process depend only on the sequence of messages it has received.

Having defined the model and analyized it, we have now refined our intuition. Notice that is so doing we not only learned about this particular problem but also about variations. For example, we might wonder whether coordination of two processes is possible if the channel never fails (so messages are never lost) or if the channel informs the sender whenever a message is lost. For each modification, we can determine whether the change invalidates some assertion being made in the analysis (i.e. the proof above) and thus we can determine whether the change invalidates the proof.

3. Synchronous versus Asynchronous Systems

When modeling distributed systems, it is useful to distinguish between *asynchronous* and *synchronous* systems. With an asynchronous system, we make no assumptions about process execution speeds and/or message delivery delays; with a synchronous system, we do make assumptions about these parameters. In particular, the relative speeds of processes in a synchronous system is assumed to be bounded, as are any delays associated with communications channels.

Postulating that a system is asynchronous is a non-assumption. Every system is asynchronous. Even a system in which processes run in lock step and message delivery is instantaneous satisfies the definition of an asynchronous system (as well as that of a synchronous system). Because all systems are asynchronous, a protocol designed for use in an asynchronous system can be used in any distributed system. This is a compelling argument for studying asynchronous systems.

In theory, any system that employs reasonable schedulers can be regarded as being synchronous, because there are then (trivial) bounds on the relative speeds of processes and channels. This, however, is not a useful way to view a distributed system. Protocols that assume the system is synchronous exhibit performance degradation as the ratios of the various process speeds and delivery delays increase. Reasonable throughput can be attained by these protocols only when processes execute at about the same speed and delivery delays are not too large. Thus, there is no value to considering a system as being synchronous unless the relative execution speeds of processes and channel delays are close.

In practice, then, postulating that a system is synchronous constrains how processes and communications channels are implemented. The scheduler that multiplexes processors must not violate the constraints on process execution speeds. If long-haul networks are employed for communications, then queuing delays, unpredictable routings, and retransmission due to errors must not be allowed to violate the constraints on channel delays. On the other hand, asserting that the relative speeds of processes is bounded is equivalent to assuming that all processors in the system have access to approximately rate-synchronized real-time clocks. This is because either one can be used to implement the other. Thus, timeouts and other time-based protocol techniques are possible only when a system is synchronous.

An Election Protocol

In asserting that a system is synchronous, we rule out certain system behaviors. This, in turn, enables us to employ simpler or cheaper protocols than would be required to solve the same problem in an asynchronous system (where these behaviors are possible). An example is the following election problem.

> **Election Problem**. A set of processes P_1, P_2, ..., P_n must select a leader. Each process P_i has a unique identifier $uid(i)$. Devise a protocol so that all of the processes learn the identity of the leader. Assume all processes start executing at the same time and that all communi-

cate using broadcasts that are reliable. ■

Solving this problem in an asynchronous system is not difficult, but somewhat expensive. Each process P_i broadcasts $\langle i, uid(i) \rangle$. Every process will eventually receive these broadcasts, so each can independently "elect" the P_i for which $uid(i)$ is smallest. Notice that n broadcasts are required for an election.

In a synchronous system, it is possible to solve the Election Problem with only a single broadcast. Let τ be a known constant bigger than the largest message delivery delay plus the largest difference that can be observed at any instant by reading clocks at two arbitrary processors. Now, it suffices for each process P_i to wait until either (i) it receives a broadcast or (ii) $\tau * uid(i)$ seconds elapse on its clock at which time it broadcasts $\langle i \rangle$. The first process that makes a broadcast is elected.

We have illustrated that by restricting consideration to synchronous systems, time can be used to good advantage in coordinating processes. The act of *not* sending a message can convey information to processes. This technique is used, for example, by processes in the synchronous election protocol above to infer values of $uid(i)$ that are held by no process.

4. Failure Models

A variety of failure models have been proposed in connection with distributed systems. All are based on assigning responsibility for faulty behavior to the system's components—processors and communications channels. It is faulty components that we count, not occurrences of faulty behavior. And, we speak of a system being t-fault tolerant when that system will continue satisfying its specification provided that no more than t of its components are faulty.

By way of contrast, in classical work on fault-tolerant computing systems, it is the occurrences of faulty behavior that are counted. Statistical measures of reliability and availability, like MTBF (mean-time-between-failures) and probability of failure over a given interval, are deduced from estimates of elapsed time between fault occurrences. Such characterizations are important to users of a system, but there are real advantages to describing the fault tolerance of a system in terms of the maximum number of faulty components that can be tolerated over some interval of interest. Asserting that a system is t-fault tolerant is a measure of the fault tolerance supported by the system's architecture, in contrast to fault tolerance achieved simply by using reliable components.

Fault tolerance of a system will depend on the reliability of the components used in constructing that system—in particular, the probability that there will be more than t failures during the operating interval of interest. Thus, once t has been chosen, it is not difficult to derive the more traditional statistical measures of reliability. We simply compute the probabilities of having various configurations of 0 through t faulty components. So, no expressive power is lost by counting faulty components—as we do—rather than counting fault occurrences.

Some care is required in defining failure models, however, when it is the faulty components that are being counted. For example, consider a fault that leads to a message loss. We could attribute this fault to the sender, the receiver, or the channel. Message loss due to signal corruption from electrical noise should be blamed on the channel, but message loss due to buffer overflow at the receiver should be blamed on the receiver. Moreover, since replication is the only way to tolerate faulty components, the architecture and cost of implementing a t-fault tolerant system very much depends on exactly how fault occurrences are being attributed to components. Incorrect attribution leads to an inaccurate distributed system model; erroneous conclusions about system architecture are sure to follow.

A faulty component exhibits behavior consistent with some failure model being assumed. Failure models commonly found in the distributed systems literature include:

Failstop. A processor fails by halting. Once it halts, the processor remains in that state. The fact that a processor has failed is detectable by other processors [S84].

Crash. A processor fails by halting. Once it halts, the processor remains in that state. The fact that a processor has failed may not be detectable by other processors [F83].

Crash+Link. A processor fails by halting. Once it halts, the processor remains in that state. A link fails by losing some messages, but does not delay, duplicate, or corrupt messages [BMST92].

Receive-Omission. A processor fails by receiving only a subset of the messages that have been sent to it or by halting and remaining halted [PT86].

Send-Omission. A processor fails by transmitting only a subset of the messages that it actually attempts to send or by halting and remaining halted [H84].

General Omission. A processor fails by receiving only a subset of the messages that have been sent to it, by transmitting only a subset of the messages that it actually attempts send, and/or by halting and remaining halted [PT86].

Byzantine Failures. A processor fails by exhibiting arbitrary behavior [LSP82].

Failstop failures are the least disruptive, because processors never perform erroneous actions and failures are detectable. Other processors can safely perform actions on behalf of a faulty failstop processor.

Unless the system is synchronous, it is not possible to distinguish between a processor that is executing very slowly and one that has halted due to a crash failure. Yet, the ability to make this distinction can be important. A processor that has crashed can take no further action, but a processor that is merely slow can. Other processors can safely perform actions on behalf of a crashed processor, but not on behalf of a slow one, because subsequent actions by the slow processor might not be consistent with actions performed on its behalf by others. Thus, crash failures in asynchronous systems are harder to deal with than failstop failures. In synchronous systems, however, the crash and failstop models are equivalent.

The next four failure models—Crash+Link, Receive-Omission, Send-Omission, and General Omission—all deal with message loss, each modeling a different cause for the loss and attributing the loss to a different component. Finally, Byzantine failures are the most disruptive. A system that can tolerate Byzantine failures can tolerate anything.

The extremes of our spectrum of models—failstop and Byzantine—are not controversial, but there can be debate about the other models. Why not define a failure model corresponding to memory disruptions or misbehavior of the processor's arithmetic-logic unit (ALU)? The first reason brings us back to the two fundamental questions of Section 2. The feasibility and cost of solving certain fundamental problems is known to differ across the seven failure models enumerated above. (We return to this point below.) A second reason that these failure models are interesting is a matter of taste in abstractions. A reasonable abstraction for a processor in a distributed system is an object that sends and receives messages. The failure models given above concern ways that such an abstraction might be faulty. Failure models involving the contents of memory or the functioning of an ALU, for example, concern internal (and largely irrelevant) details of the processor abstraction. A good model encourages suppression of irrelevant details.

Fault Tolerance and Distributed Systems

As the size of a distributed system increases, so does the number of its components and, therefore, so does the probability that some component will fail. Thus, designers of distributed systems must be concerned from the outset with implementing fault tolerance. Protocols and system architectures that are not fault tolerant simply are not very useful in this setting.

The link between fault tolerance and distributed systems goes in the other direction as well. Implementing a distributed system is the only way to achieve fault tolerance. All methods for achieving fault tolerance employ replication of function using components that fail independently. In a distributed system, the physical separation and isolation of processors linked by a communications network ensures that components fail independently. Thus, achieving fault tolerance in a computing system can lead to solving problems traditionally associated with distributed computing systems.

Failures—be they hard or transient—can be detected only by replicating actions in failure-independent ways. One way to do this is by performing the action using components that are physically and electrically isolated; we call this *replication in space*. The validity of the approach follows from an empirically justified belief in the independence of failures at physically and electrically isolated devices. A second approach to replication is for a single device to repeatedly perform the action. We call this *replication in time*. Replication in time is valid only for transient failures.

If the results of performing a set of replicated actions disagree, a failure has occurred. Without making further assumptions, this is the strongest statement that can be made. In particular, if the results agree, we cannot assert that no component is faulty (and the results are correct). This is because if there are enough faulty components, all might be corrupted, yet still agree. For Byzantine failures, $t+1$-fold replication permits t-fault tolerant failure detection but not masking. This is because when there is disagreement among $t+1$ independently obtained results, one cannot assume that the majority value is correct. In order to implement t-fault tolerant masking, $2t+1$-fold replication is needed, since then as many as t values can be faulty without causing the majority value to be faulty. At the other extreme of our failure models spectrum, for failstop failures a single component suffices for detection. And, $t+1$-fold replication is sufficient for masking the failure of as many as t faulty components.

5. Which Model When?

Theoreticians have good reason to study all of the models we have discussed. The models each idealize some dimension of real systems, and it is useful to know how each system attribute affects the feasibility or cost of solving a problem. Theoreticians also may have reason to define new models. Identifying attributes that affect the problems that arise in distributed systems allows us to identify the key dimensions of the problems we face.

The dilemma faced by practitioners is that of deciding between models when building a system. Should we assume that processes are asynchronous or synchronous, failstop or Byzantine? The answer depends on how the model is being used. One way to regard a model is as an interface definition—a set of assumptions that programmers can make about the behavior of system components. Programs are written to work correctly assuming the actual system behaves as prescribed by the model. And, when system behavior is not consistent with the model, then no guarantees can be made.

For example, a system designed assuming that Byzantine failures are possible can tolerate anything. Assuming Byzantine failures is prudent in mission critical systems, because the cost of system failure is usually great, so there is considerable incentive to reduce the risk of a system failure. For most applications, however, it suffices to assume a more benign failure model. In those rare circumstances where the system does not satisfy the model, we must be prepared for the system to violate its specification.

Large systems, especially, are rarely constructed as single monolithic entities. Rather, the system is structured by implementing abstractions. Each abstraction builds on other abstractions, providing some added functionality. Here, having a collection of models can be used to good advantage. Among the abstractions that might be implemented are processors possessing the attributes discussed above. We might start by assuming our processors only exhibit crash failures. Failstop processors might then be approximated by using timeout-based protocols. Finally, if we discover that processors do not only exhibit crash failures we might go back and add various sanity-tests to system code, causing processors to crash rather than behave in a way not permitted by the crash failure model.

Lastly, the various models can and should be regarded as limiting cases. The behavior of a real system is bounded by our models. Thus, understanding the feasibility and costs associated with solving problems in these models, can give us insight into the feasibility and cost of solving a problem in some given real system whose behavior lies between the models.

References

[BMST92] Budhiraja, N., K. Marzullo, F.B. Schneider, and S. Toueg. Primary-backup protocols: Lower bounds and optimal implementations. *Proceedings of the Third IFIP Working Conference on Dependable Computing for Critical Applications* (Mondello, Italy, Sept. 1992), 187-198.

[F83] Fischer, M.J. The consensus problem in unreliable distributed systems (A brief survey). *Proc. 1983 International Conference on Foundations of Computations Theory, Lecture Notes in Computer Science*, Vol. 158, Springer-Verlag, New York, 1983, 127-140.

[H84] Hadzilacos, V. *Issues of Fault Tolerance in Concurrent Computations* Ph.D. thesis, Harvard University, June 1984.

[LSP82] Lamport, L., R. Shostak, and M. Pease. The Byzantine generals problem. *ACM TOPLAS 4*, 3 (July 1982), 382-401.

[PT86] Perry, K.J. and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering SE-12*, No. 3, (March 1986) 477-482

[S84] Schneider, F.B. Byzantine generals in action: Implementing fail-stop processors. *ACM TOCS 2*, 2 (May 1984), 145-154.