1983 Invited Address

Solved Problems, Unsolved Problems and Non-Problems in Concurrency

Leslie Lamport[1]

This is an edited transcript of a talk given at last year's conference. To preserve the flavor of the talk and the questions, I have done very little editing—mostly eliminating superfluous words and phrases, correcting especially atrocious grammar, and making the obvious changes needed when replacing slides by figures. The tape recorder was not functioning for the first few minutes, so I had to recreate the beginning of the talk.

It's an honor to be invited to speak here at the second PODC conference. I'd like to think that it was because of my work, but I suspect that the real reason is that I tend to be controversial, saying all sorts of things that will offend people and liven things up. To paraphrase Isaac Newton:

> *If I have received more notice than other men, it was by stepping on the toes of giants.*

Well, I'll try not to disappoint you.

What I'm going to present here are my own personal views. Needless to say, I don't expect most of you to agree with these views. Saying what is and is not a problem involves prediction—deciding just what will be regarded in the future as the real problems in concurrency. To show how successful I am, I'll just tell you that when Susan Owicki first described Pnueli's use of temporal logic—in a seminar she gave around the summer of '79 or '80—I knew immediately that it was formal nonsense that wasn't really good for anything.[2] So, with that piece of prognostication to vouch for my abilities as a fortune teller, I'll begin.

[1]Work supported in part by the National Science Foundation under grant number MCS-8104459, and by the Army Research Office under grant number DAAG29-83-K-0119.

[2]After the talk, I was told that some people didn't get the joke and thought I was insulting Pnueli. In fact, much of my recent research has been based upon Pnueli's work in temporal logic.

Shared Variables	1965
Semaphores	1968
Monitors	1972
CSP	1978

Figure 1: The history of standard concurrency.

The first unsolved problem I want to talk about is the problem of developing a fundamental theory of concurrency. By a fundamental theory, I mean one that's not based upon arbitrary formal models or specific languages, but one that's really fundamental.

I recently heard a speaker use the phrase "standard concurrency". After listening to him for a while, it turned out that by "standard concurrency" he meant CSP.

> *Those who cannot remember the past are condemned to repeat it.*
>
> *George Santayana*

The history of standard concurrency is shown in Figure 1. When I started working in concurrency, standard concurrency meant semaphores. (Actually, conditional critical regions started challenging semaphores, but monitors became the standard before conditional critical regions had a chance.) Papers that claimed to be about synchronization were really about semaphores.

Well, synchronization isn't really semaphores, or monitors or CSP. Synchronization is something more fundamental. I don't mean to put down CSP. It's a fine language—or, more precisely, a fine set of communication constructs. Hoare deserved his Turing award. But, so did Dijkstra.

Some of you may think that was way back then, but now we *really* know what concurrency is all about, and we really know that CSP is the right way of doing things. For those of you who think that way, I'd like to remind you that while we theoreticians are busy studying CSP, people out there in the real world are building Ethernets. And CSP doesn't seem to me to be a very good model of Ethernets.

Anyway, what I really want to talk about is a fundamental theory of synchronization or concurrency. To give you an example of what it is that I would like to see, I'll talk about something that I do understand a little about—

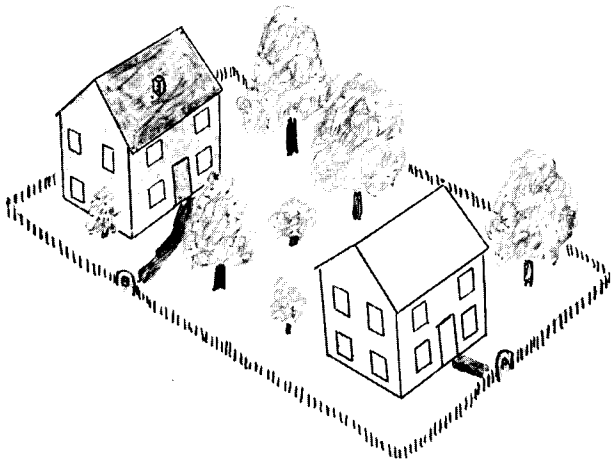

Figure 2: Alice and Bob's yard.

namely, the mutual exclusion problem.

In the mutual exclusion problem, we have two people—let's call them Alice and Bob. Alice and Bob are neighbors and share a yard, as shown in Figure 2. Alice and Bob also have dogs, and naturally they want to let the dogs use the yard. After all, it's a lot easier just to let the dog out in the yard than to go out and walk him. The problem is that these dogs don't like each other, and they fight, so only one dog at a time can be in the yard. As you can see from the picture, there are trees and bushes in the yard, so you can't just look out the window and see if there's a dog in the yard. So, Alice and Bob had to do something to communicate with each other to find out who can let his or her dog into the yard when.

They could just walk over to the other person's house and knock on the door, but they didn't want to do that. After all, it might be raining, and they wanted to stay nice and dry and comfortable in the house. So, they needed some way of communicating. How did they do it?

The first idea Alice had was to get some walkie-talkies. Then they could just sit in the house and call each other to decide whether it was safe to let a dog out. Unfortunately, that doesn't work. The problem is that in order for a walkie-talkie to work, you have to keep the walkie-talkie with you at all times—or at least while your dog is in the yard. But what happens if Alice, say, puts the dog in the yard and then wants to take a shower or run down to the store for something. She'd have to keep the walkie-talkie with her at all times, and that won't do.

What they needed was a less ephemeral sort of message-passing device, so Bob had a really clever idea. He devised the little device, shown in Figure 3, for transmitting information. The can sits on Alice's window sill, and the string runs over to Bob's house. When Bob wants to send one bit of information to Alice, he just pulls on the string and knocks the can down. Alice can look at any time and see if the can is up or down.

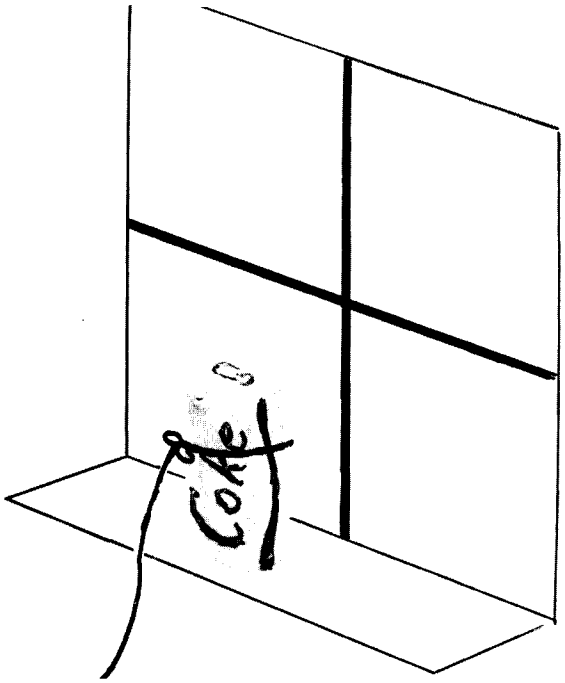

Figure 3: A simple signaling device.

Well that seems really neat, but unfortunately, that doesn't work either. The problem is that you can only put a finite number of cans on the window sill. This means that Alice, say, can let her dog into the yard only a fixed number of times before Bob has to reset the cans. In other words, each time Alice wants to let the dog out, she has to send some message, pulling down at least one can. With a finite number of cans, she can only do that a finite number of times before Bob has to do something about resetting the cans. But what happens if Bob goes on vacation without telling Alice? He could be gone for weeks, and Alice is bound to run out of cans. So this doesn't work.

Finally, they consulted a computer scientist, who tells them: "Yes, we know this problem. For signalling each other, all programmers know you need some kind of flag—a binary device which can be set and reset by one person." So, they went home and built some flags, like the one shown in Figure 4. Each of them then had a flag, and they could just reach out the window and raise or lower their flags. And, in fact, with these flags, there's an algorithm that solves their problem. I'll show it to you in a minute.

Now what have I just told you? I claim that I've told you some very important fundamental properties of mutual exclusion. The first thing I told you is that mutual exclusion is not solvable by message passing—at least not the way the mutual exclusion problem is normally formulated. If you look at solutions which claim to implement mutual exclusion with message passing, you realize that they are assuming some lower-level mechanism that actually implements the mutual exclusion. Usually, you think you're communicating by message passing; but what the

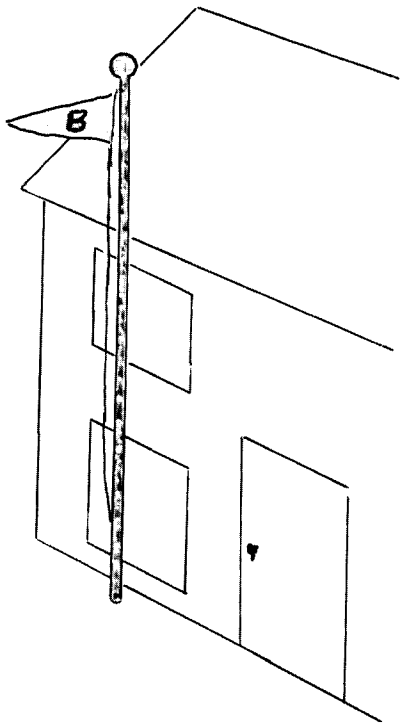

Figure 4: A one-bit flag.

message is really doing is setting a flag at the other end, and that flag is logically part of the process that is sending the message.

The other thing I told you was that the mutual exclusion problem is not solvable with interrupts. Those tin cans I drew were really the communication mechanism that's more familiar to you as an interrupt bit—it's set by the sender and it's reset by the receiver. It can't be used to solve the mutual exclusion problem, as that problem is usually posed.

But the mutual exclusion problem *is* solvable with one-bit registers that have one reader and one writer—with no lower-level mechanism.

Notice I told you all this without constructing any models, without constructing any formal theories, without having to define any formal programming language. The point I'd like to make is that synchronization is a real physical property; it's not dependent on models. A theory of synchronization should in some sense be a theory of physics. If someone asks "What's the mutual exclusion problem?" we say: "It's that two guys can't be at the same place at the same time." Place and time are physical concepts. Even though we may build some models to help us understand the problems, remember that the problems we are trying to solve have to do with real computers moving real electrons around.

I noticed—at least when I started in this game, it's been less true recently—that people tend to approach these problems by building models. The first thing someone would

```
A: while true
       do F_A := up;
          while F_B = up do skip od;
          dog in yard;
          F_A := down
       od

B: while true
       do F_A := up;
          while F_B = up do F_B := down
                                   while F_B = up do skip od;
                                   F_A := up;
                               od;
          dog in yard;
          F_A := down
       od
```

Figure 5: A mutual exclusion algorithm.

ask when I said I'd like to talk about synchronization is: "What's your model?" What I'm saying is: Let's think about real results. Models come after you start understanding things.

I mentioned that there was an algorithm for solving the mutual exclusion problem. It's the one in Figure 5. I've written it as a nice little program with **while**s and shared variables and everything. It's a fairly well-known algorithm. What's less well known about this algorithm is that it works with truly concurrent reads and writes. No assumption of atomicity is needed anywhere along the line. Incidentally, it's a chivalrous algorithm; Alice has priority over Bob.

Now let me show you the same algorithm written in a slightly different way. In Figure 6, I've written it as a CSP program. It's more or less the same algorithm, but I should mention that it's not the identical algorithm. The way I've written it in CSP states that the operations of reading and writing the flag are atomic, so it's not as general as the algorithm shown in Figure 5. You can write the other algorithm in CSP, but it would be hard to fit it on a single slide. CSP isn't a very good language for describing this kind of algorithm, although it's good for other kinds of algorithms.

Now, is this a distributed algorithm? Well, when I showed you the picture of the yard (Figure 2), it looked pretty distributed. There were two people in different houses communicating by signalling over a distance. When I write it as Figure 6, we have a nice distributed CSP algorithm. But if I just wrote it as Figure 5 and didn't give you these clues, you'd say that it's just an ordinary shared-variable nondistributed program.

Which is it? Is it distributed or is it not distributed? Well, we've come to our first nonproblem: *What is a distributed system?* Distribution is in the eye of the beholder. To the user sitting at the keyboard, his IBM personal computer is a nondistributed system. To a flea crawling around

```
A:: *[true → F_A ! up;
          F_B ? t;
          *[t = up → F_B ? t];
          dog in yard;
          F_A ! down          ]

B:: *[true → F_B ! up;
          F_B ? t;
          *[t = up → F_B ! down;
                     *[t = up → F_A ? t ];
                     F_B ! up;
                     F_A ? t;              ];
          dog in yard;
          F_A ! down                       ]

F_A:: *[F_A ? val → skip □
        F_B ! val → skip     ]

F_B:: *[F_B ? val → skip □
        F_A ! val → skip     ]
```

Figure 6: The algorithm rewritten in CSP.

on the circuit board, or to the engineer who designed it, it's very much a distributed system.

When people ask "Is X a distributed system?", I think they're really asking a question of morality. We know that distributed systems are good and nondistributed systems are bad. I think I understand why: The Defense Department gives money to study distributed systems and doesn't give money to study nondistributed systems, so that means distributed systems are good.

Another nonproblem: *Distributed proofs of distributed algorithms.* You just saw me describe the same algorithm two different ways. One is a "distributed" CSP program, the other is a "nondistributed" shared-memory program. I think we're in trouble if the way we prove this algorithm depends on which way we represent it, because then our proofs aren't telling us very much about the algorithm, they're telling us about the programming language we happen to write it in. So, this notion of trying to prove distributed algorithms differently from nondistributed algorithms is a nonproblem.

The source of the confusion here is that people confuse distribution with modularity. We know that modularity is a good thing. Modularity is important for a proof—we want to break our proof up into pieces, that is, into separate *modules.* But the notion of a module and the notion of a process are orthogonal concepts. They don't have anything to do with one another.

Let's take an example. Suppose we have a distributed file system in which there are computers in New York and San Francisco, like the one in Figure 7, and suppose we have two programmers assigned the job of implementing

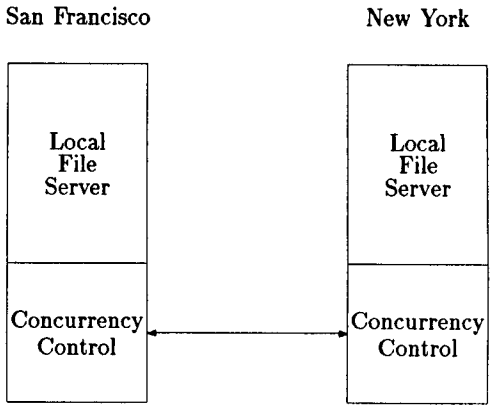

Figure 7: A two-process system.

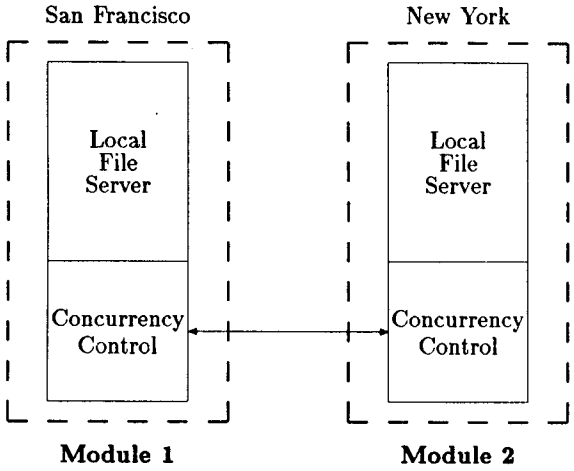

Figure 8: The obvious decomposition.

it. We have a nice distributed two process system—one process in New York, one process in San Francisco—and we want to decompose it into two modules, one for each programmer. Of course, everybody knows an obvious way of doing this: the one shown in Figure 8. You have one programmer implementing the module in San Francisco, and the other programmer implementing the one in New York. When they're done, they come together in Chicago and try to put it together. Right? Of course not! The way you break this up into two modules is shown in Figure 9, where each module involves parts of both processes.

Similarly you don't try to prove the correctness of a two-process algorithm by writing proofs of each module separately and then pasting them together. You write a proof of an algorithm by viewing what the algorithm does, and that generally involves global reasoning. If the two processes are intimately connected and intimately communicating, you don't go off and write proofs of each one separately and then try to paste the proofs together.

So we come to a solved problem, which is: *Proving*

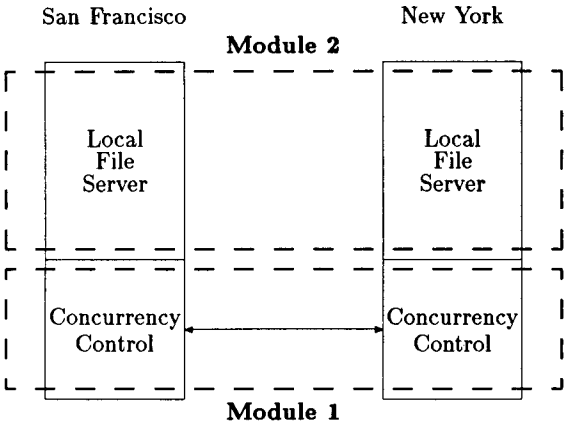

Figure 9: A better decomposition.

properties of concurrent programs. There are two kinds of properties that we prove. The first is *safety properties.* The method that's known for doing this is generally called the Gries-Owicki or Owicki-Gries method, but I like to attribute it to Ashcroft who was really the first one who did it—in 1972. My favorite formulation of it, which I think makes things clear, is what I call the Generalized Hoare Logic. It appeared in my 1981 *Acta Informatica* paper, and it is explained much more clearly in a forthcoming paper by Fred Schneider and myself.[3] Basically, this is a generalization of the Hoare partial correctness approach, except that instead of pre- and post-conditions you have invariance.

The other kind of property is *liveness.* We know how to prove those properties now. The proofs involve making lattices of temporal assertions and using well-foundedness arguments. The basic idea for that came from Pnueli's work on temporal logic. What's really new—what makes concurrent programs different from sequential programs—is that the assertions you put into the lattice are temporal logic assertions rather than ordinary logic assertions. Susan Owicki and I wrote a paper that tells you how to do it; it appeared in *TOPLAS* in 1982.

These methods work just as well for distributed programs as for nondistributed programs. There's nothing special about distributed programs. There's still work to be done in program verification. There are new language constructs—we want to find appropriate proof rules for them—and there's a lot of other stuff to be done. But you're not going to discover any remarkable, brilliant new method of verifying concurrent programs—just as you're not going to discover any remarkable, brilliant new method of verifying sequential programs. Axiomatic methods for sequential programs are going to boil down to Hoare pre-/post-conditions for proving safety properties and some well-foundedness argument for proving termination. The same thing is going to be true of concurrent programs. The fantastic new method that you may come up with is going to be just a reformulation of these basic ideas in some new, confusing method.

[3]This paper has since appeared in the April, 1984 issue of *TOPLAS.*

Well, what are the unsolved problems of program verification? What I regard as the main unsolved problem is: *Proving properties of* real *programs.* I bank at the Bank of America. They have an automated-teller card, called a Versatel card. So, I go down to my friendly automated teller, where I put in my card and say "Give me some money." Usually it does. However, every so often, the teller comes back and says: "I'm sorry; system not available," or something like that, which is a euphemism for "I'm sorry, things have crashed." Usually it's a communication problem; the teller can't communicate with the central computer. But sometimes it's the central computer that has crashed.

If you look at when the Bank of America central computer crashes and start doing statistics, you find that occasionally somebody has spilled coffee in the mainframe, or something like that happened that you can understand. But, in about 95% of those crashes, they don't really know what caused it. The system goes down, somebody says "Oh, here it goes again," and they push the reboot button. It takes them a couple of minutes to bring the stuff in from disk, the system starts again, and everying is fine. So, depending on the particular prejudices of whoever is recording those things, it was either a transient hardware failure, or whatever they like to call it. But basically, they just don't really know what happened.

My guess is that most of those crashes are concurrent programming errors, because they are precisely the sort of things you expect when you have synchronization errors. These failures occur occasionally, you can't detect them, you can't get rid of them by testing, you can't reproduce them—they sound just like concurrent programming software errors.

What the world needs are not more proofs of ten-line concurrent algorithms. The world needs some way of getting Bank of America to be able to eliminate those 95% of their crashes—some tool, some method, maybe some way of teaching programmers how to use the techniques that we already have, but some way of getting these proof methods out into the real world. I strongly advise people to knock on the doors of Bank of America and say. "Hey, can we help you?"

A similar current research area—I wouldn't exactly call it an unsolved problem—which I think is very important is: *High-level specification and proof.* I just told you that we know how to prove concurrent programs, so why do I now have a current research area called " high-level specification and proof"? Well, in the standard verification methods like the Owicki-Gries method, what you're proving are statements about the program written in the language of the program. You prove statements about program variables, about bits, and about all the stuff that the program is shuffling around. But when you write a specification of, say, what the Bank of America's Versateller is supposed to be doing, you should write it in terms of how the customer

If *request A* $\rightarrow$ *request B*
then *service A* $\rightarrow$ *service B*

Figure 10: An informal specification of FIFO.

looks at it. The customer isn't interested in bits and flags and fields and such; he's interested in those little green pieces of paper coming out of the machine. So, if you want to write a specification for the Bank of America system, you want to talk in terms of those little green pieces of paper and the buttons that the user hits, not in terms of bits and bytes and nibbles and all that stuff. The specification should be independent of a particular implementation.

We know how to verify properties that involve the actual bits of the program, but when you write those properties as high-level statements, not involving bits but involving transactions, there are still unsolved problems involved in proving that the program, which shuffles bits and bytes, is really doing the right thing when viewed at the level of button-pushing and dollars coming out. The important aspect of the specification is trying to specify the problem not the solution. As I've indicated, temporal logic seems to be a promising tool—despite my initial reaction to it—but it's not the only approach that's being used. I regard all this as a very fertile field of research.

To illustrate one particular unsolved problem in specification, I'll mention that the problem of specifying priority is completely unsolved.[4] For example, I'll talk about specifying FIFO (first-in, first-out), which is a simple type of priority. Now at this point everybody who does specifications (Amy[5] for example is here someplace) will have their back bristling and will be saying: "What do you mean FIFO can't be specified? Anybody can specify that—especially anybody using temporal logic." You can write down this nice temporal logic expression, and obviously that means FIFO.

Let's look a little more closely at a FIFO specification, which I've written in Figure 10, where the $\rightarrow$ is read as "precedes". This specification says: For all A and B, if *request A* precedes *request B*, then *service A* has to proceed *service B*. In other words, if A requests service before B does, then A should get service before B does.

Whenever you write a specification, you have to say something about what the interface operations are. I could talk a lot about that; it's an area of specification that people tend to neglect. If you look at most high-level specifications, you'll find out that you can't tell from the specification whether what's being specified is a Pascal program or a box with a bunch of wires sticking out of it. Somehow, I would say you haven't really specified something completely if you don't know whether you're supposed to get a Pascal program or a box with a bunch of wires sticking out. The reason you don't know that is that you don't know what the *interface* is. It's the interface that's different for those two cases, and you don't know whether the interface involves Pascal subroutine calls or electronic voltage levels.

In writing a FIFO specification, you usually assume operations of *enter request* and *give service*. So, when we write the specification, we assume that we have these operations—the operation in which A can enter a request and the one in which A can receive service. These operations have to be implementation-dependent; you have to specify them very differently for a Pascal program than for a box with wires sticking out of it. For a Pascal program, the operation of entering a request might consist of calling a certain subroutine, and the operation of granting the request might simply be the return from that subroutine.

Notice that we have *request* as the operation in the specification, whereas *enter request* is the operation that's being assumed. What's the relation between the two? There are two possibilities. The first possibility is that the *request* should be synonymous with the *enter request*, so that when we write that *request A* precedes *request B*, we really mean that *enter request A* precedes *enter request B*.

You can write that as your specification, and it looks very nice. The problem is that it's not always implementable. For example, if your interface is in a machine language and *enter request* is simply branching to a certain location, then there is no way a computer can figure out which of two processes branched to that location first. So, this is a nice-looking specification, but it's not always implementable.

So much for the first possibility. The second possibility is that the *enter request* is just a part of the *request* operation. Another way of saying that is that *request* is an operation which the implementer is free to define. While some things are specified as are part of the interface, there are other things that the implementer is allowed to define. For example, if a specification is in terms of a queue, the implementer is free to define exactly how that queue is represented. In the same way, you could say that the implementer is free to define exactly how the operation of requesting something is implemented.

That's fine, but the problem is that once you've done this, any algorithm is FIFO. All you have to do is define the *request* operation to be everything that happens from the time the user says *enter request* until he's actually granted that request. You then discover from this definition that every algorithm is FIFO.

If you start thinking now about what you really mean by FIFO, you'll notice that there's an implicit condition that the *request* operation must not involve any waiting. That's why the definition in which the *request* operation is everything from the time you enter the request until the time it's granted doesn't work, because that part of the implementation involves waiting for the other process to do something. Unfortunately, the notion of "no waiting" isn't expressible in any methodology I know of. So, here we have the simplest type of specification you might think of

[4] This problem was discovered during a discussion with Richard Schwartz and Michael Melliar-Smith.

[5] Amy Lansky

in a concurrent system—a FIFO protocol—and no current specification method can specify it.

Okay, let's get back to fundamental theory. Let's go back to Alice and Bob and their yard (Figure 2). After Alice and Bob had used their protocol for a while, their dogs started becoming friends and the whole business wasn't necessary. So, they took down their flags. Then Bob and Alice also got friendly, and they got married. But being a modern California fairy tale, they soon got divorced. Part of the divorce settlement was that Alice had custody of the dogs and Bob had to pay canine support, which in this case meant providing dog food. Now, there was a very bitter custody battle for the dogs, so Alice and Bob didn't want to see each other. The arrangement was that Bob would put dog food in the yard when Alice wasn't there and Alice would go get the food when Bob wasn't there. Alice would feed the dogs and, when the dog food ran out, she would tell Bob that she needed more dog food, and Bob would put it in the yard.

As you probably know, what I've just described is a producer-consumer problem. On the face of it, it looks very much like a mutual exclusion problem because you have the same condition: Bob and Alice aren't supposed to be in the yard at the same time, just as in the mutual exclusion problem their dogs weren't supposed to be in the yard at the same time. But in fact, it's a very different problem. Why is it different? First of all, you can solve it without the flags; you can use the cans (interrupt registers) to solve it. This shouldn't come as a surprise to anyone, but most people never really thought about it. When you have a main frame and a peripheral unit, they're solving a producer-consumer problem between them, and they do it communicating through an interrupt register.

The other difference, which I regard as more fundamental, is the fact that there is no arbiter involved in the producer-consumer problem, whereas there is one in the mutual exclusion problem. Okay, what's an arbiter? An arbiter is a device that makes a discrete decision based on continuous input—for example, the flag position. When you raise the flag, there are an infinite number of positions it can be in, not just up or down. But when Alice reads the flag—when she says to herself: "Is that flag up?"—she has to take those infinite number of possibilities and make a discrete decision: up or down. A device that does this is known as an arbiter.

It appears to be a law of physics—not a law of somebody's model of concurrency, but a real law of physics—that it is impossible to bound the arbiter's decision time. In other words, there have to be situations— some flag position or some particular circumstance of raising the flag—in which it will take Alice 100 years to decide:

> Is it up? Is it down? Well ... no, it's up. Is it really up? No, I think it's ... But maybe it's d... No, it's ... But maybe ...

The probability of that going on for too long is pretty small, but it appears to be a fact that the possibility of such a delay is inevitable.

It's a very interesting problem. Physicists don't seem to be aware of it, but I regard the arbiter problem as one of the interesting facets of concurrency.

So, an important difference between the mutual exclusion problem and the bounded buffer or producer-consumer problem is that any algorithm that solves the mutual exclusion problem cannot be guaranteed to work in a bounded length of time, whereas the producer-consumer problem does have solutions that take a bounded length of time.

Again, I didn't say anything about models; I'm talking here about the laws of nature. That's the sort of thing I'd like to see in a theory of concurrency. I'd like to see this kind of result somehow codified and made more precise.

What are the unsolved problems here? I told you some things I know about it; what don't I know? Here's one unsolved problem. I mentioned two types of synchronization: mutual exclusion and producer-consumer. Are they the only kinds of synchronization problems? Can any problem be regarded as a mutual exclusion problem or a producer-consumer problem, or a combination of the two? Every problem I've seen can be; but I wouldn't know how to go about saying that this is true or false, or exactly what it means. I think there's an interesting problem here.

Another problem is: *What are the fundamental costs of synchronization?* There have been a few results. Nancy Lynch and Mike Fischer, among others, have been working on this, but we don't even really know too much about how to measure cost. Should we count the number of messages? The number of message delays? If you're dealing with a shared memory situation, are these reasonable things to measure? Message passing in some sense is really equivalent to shared variables if you look at it at a lower level. So, what does this all mean? I don't know.

Another interesting problem is: *What are the fundamental limits to concurrency?* The number of processors is usually considered the limitation; give me enough processors and I'll compute anything for you. Everybody talks about these wonderful ways of putting processors together, and everybody is building nice systolic arrays—regular arrays of processors. We all know that's good engineering practice because everything is simple and nice and neat. The problem is that I know one firm that's been designing some very good concurrent computers for quite a while. I don't know exactly how they work—I've never really learned too much about them—but I do know some of the other products they design have been engineered quite well. I suspect this company knows what's it doing, only when you look inside their computer you don't see this nice regular array of things; you see an awful rat's nest of processors interconnected to one another—very bad engineering. Something must be wrong with that firm, but they seem to be doing pretty well. Figure 11 shows an example of their top-of-the-line model; it's not at all the way people think of designing computers. But an organization that's been around for half a billion to a billion years must be

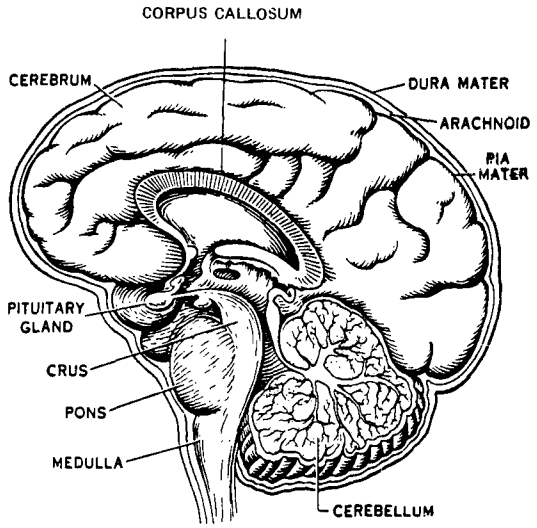

Figure 11: A well-known company's top-of-the line concurrent computer.[6]

doing something right. Which leads me to suspect there's a reason why they don't have these nice regular arrays of processors and use this rat's nest, in which neurons are connected to thousands of other neurons.

It seems clear that when you start looking at the real limits of concurrency, you will find that interconnectivity of processors is going to be more important than just how many processors you can squeeze on to your chip. Can you quantify that? Is there some theory here?

These are just a few of the problems that occurred to me when I was putting together this talk. It's certainly not a complete list. Again, I want to emphasize that I'm not looking for models, I'm looking for results. You should really understand the difference between a model and reality. I recommend looking at Turing's original paper on Turing machines. Look at how he justified his definition of the machine as a universal computer. It wasn't in terms of five-tuples of this, that and the other thing.

Okay, let's switch topics: Fault tolerance. Those of you who regard fault tolerance as one of your areas of interest, please raise your hands.[7] Now, those of you who have your hands up, keep them up if you've read this paper:

> E. W. Dijkstra: "Self-stabilizing Systems in Spite of Distributed Control." *Communications of the ACM 17*, 11 (November 1974), 643-644.

If you haven't read this paper, put your hands down.

I think most of the hands went down. Most people haven't seen this paper. Well, it *is* by an obscure author in an obscure journal, so I suppose you should be excused.

I regard this as Dijkstra's most brilliant work—at least, his most brilliant published paper. It's almost completely unknown. I regard it to be a milestone in work on fault tolerance. The terms "fault tolerance" and "reliability" never appear in this paper. In fact, one reason why it's not better known might be Dijkstra's approach, illustrated by this quote from the paper:

> The appreciation [of these results] is left as an exercise for the reader.

I regard self-stabilization to be a very important concept in fault tolerance, and to be a very fertile field for research. I'll let you dig out Dijkstra's paper and discover for yourself what self-stabilization means, and leave it as an exercise for you to appreciate what it has to do with fault tolerance. Maybe I've given you a big enough hint so that you will not, like most people at the time, just dismiss it as an interesting little exercise that's not really worth anything.

Now for the last unsolved problem that I'll mention. Since about a third of the papers in this conference seem to be fault-tolerant algorithms or Byzantine Generals sort of things, I thought I'd give at least one unsolved problem in fault tolerance:

> *Prove that clock synchronization in the face of a single arbitrary failure is impossible for a three-process system.*

I find it interesting that the analogous result for the Byzantine Generals problem was one of the first ones known, but nobody has yet proved it about clock synchronization.[8] The approximate Byzantine Generals results show that none of the straightforward ways will work—a straightforward way being one where everybody reads everybody else's clock, gets a bunch of numbers that are approximately the same, and then does something with those numbers. But nobody has ever proved that there isn't some really weird method that doesn't do it this way and somehow works. No one believes it—at least no one who understands the Byzantine Generals impossibility results—but nobody has proved it. It's a nontrivial problem and I recommend it as being more worthwhile than the latest little wrinkle in what you can or cannot do in terms of defining variants of the Byzantine Generals problem. This clock synchronization question is a very fundamental problem.

Another fundamental question is: *What's the significance of the number 4?* This may not be meaningful to many of you; to those of you to whom it isn't, I will explain that 4 is in fact equal to 3+1. What's significant here is that Byzantine Generals problems can't be solved with three processors in the face of a single arbitrary fault, but they can be solved with four processors. The reason I mention this as a problem is that there's a clock synchronization algorithm that we know about, which is called interactive convergence, which also has the same property; it works with 4 processors but not for 3 processors.[9] But the way

[6] Reproduced from *Stedman's Medical Dictionary, 21st Edition*, ©1966, the Williams & Wilkins Company, Baltimore, Md.

[7] A large portion of the audience raised their hands.

[8] This lecture stimulated Dolev, Halpern and Strong to solve the problem. Their proof appears in the *Proceedings of the Sixteenth Annual ACM STOC Conference* (1984).

[9] This algorithm is described in the paper by Melliar-Smith and me in

the number four pops up there is really strange—very differently from the way it comes up in Byzantine Generals types of solutions. Whenever I see the same number popping up in two very different ways in related areas, I think that there may be something fundamental going on, so I pose this as my last general question.

Okay, I've tried to leave plenty time for questions or attacks or whatever you wish to call them.

QUESTIONS

The identity of questioners is given when known.

Albert Meyer: One thing that I'm a little mixed up about is your objection to models. You can't really mean that you want no models. I presume this because you ask, as a set of open problems, to make precise such things as whether mutual exclusion plus producer-consumer equals synchronization. You stated a bunch of things that sounded like appeals for theorems, like that you need interrupts or you need flags, or you need arbiters, or you don't. At the same time, you're really quite negative, saying: "Enough of this models business. Distributed computation and synchronization are about 'reality'." It seems to me that those two appeals are not consistent; I presume that you mean there is a certain kind of model you don't like. But, if you're going to make things more precise in order to confirm the facts and theses that you're asserting, how else do you do it but to give some kind of precise model of what those terms mean?

Answer: As I'm fond of misquoting: "A foolish consistency is the hobgoblin of small axiom systems." What do I mean by "no models"? Take an interesting result in computability, like the halting problem. You can give a little proof of the impossibility of the halting problem without ever mentioning a five-tuple. You can talk about it in terms of drawing pictures of Turing machines and tapes, and people understand what you're doing. Any good mathematician will say: "Yes, that's really a theorem."

Of course, you might at some point want to make sure that it really does work. You might then want to formalize it and make those five-tuples and precisely define the mappings. We do make mistakes—and in the area of concurrency we make a lot more mistakes than we make in other areas—so formalism is really important. But formalism comes later. If you're doing things right, you can state and "prove" your results without ever having to get into all these epsilons and deltas and omegas and stuff. Once you've got the result and once you've explained it to other people, then, when you have to get it published in *JACM*, you can put in these epsilons and deltas and omegas. But the thing that I advise against and abhor is the notion that the first thing you do when you study a problem is sit down and find a model. We don't need that. We have reality around us; we don't need models to start with.

this proceedings.

That's what I mean by no models. Certainly, once you've got some ideas and some results, you can start formulating them more precisely in terms of these models. I always think of Einstein's remark, which I'm sure I'm misquoting, that if you really understand something you can explain it to a 13-year old child. That's the way I feel too. If you have to dive into the formal models before you can explain what you're talking about, then you're doing something wrong.

J. Misra: I was thoroughly confused when you said "model", and I think that your answer has left me even more confused. I believe if you read Dijkstra's paper, he starts out with a model—a model of process failure. Is that what you call a model, or is it against formalism that you're speaking?

Answer: Well, obviously I'm not pointing to Dijkstra's paper as a model of how to explain your ideas. I guess my comments on models are reactions to having read, way back when, several Ph. D. theses where the only difficult problem that the author solved was understanding his own notation. I think good researchers do what I'm saying automatically. They don't start with models, they start with ideas. Maybe I'm saying that bad researchers shouldn't even begin because they begin by starting with models. There's an art to doing science and mathematics, involving the fine line between rigor and formalism. I'm all in favor of rigor; I don't care too much for formalism. I can't explain where that line lies; it's something that I'm afraid isn't being taught too well in the schools. I think that if you have to ask what the difference between rigor and formalism is, then maybe you shouldn't be doing research.

Nancy Lynch: I think that part of the problem is that the models that we now have in this area aren't clean and general enough. Perhaps people can prove results in a specific model that really ought to be stated in a simpler and more general way; so we don't quite have the right way of stating our results.

Answer: I should say something in defense of the work that's been done. There's good work that is clearly saying something about the real world, not just about the specific model that it's stated in. But I think we don't understand the real world of concurrency well enough to be able to state them right. So, I don't think we have the final statements of those results. The way of improving this situation is not to look for models, but to try to study electrons and computers and what's really going on.

Michael Rabin: I would like to add a comment to this long discussion. I completely agree with your point of view that there is too much formalism. I think I would rather express your objections to some of what is going on as an objection to an excess of formalism. The difference between, on the one hand, Turing's work and the work on computability in the sense of recursive functions, and, on the other hand, the kind of thing that we were discussing here, is that fortunately the notion of computabil-

ity of a function from the integers to the integers is really rather invariant. There were several models or, if you wish, formalisms: Church's lambda calculus, Turing's, Post's, Markov's, Kleene's, etc. Fortunately they all capture a notion that is really invariant. Therefore, we have the luxury of expressing things in invariant terms and talking about the proof and not about the formalism. Programs, which are algorithms realized in computers, are not as clean—at least the way we view them now, perhaps without having the ultimate understanding of them. That, to my mind, may be the difficulty.

My second remark will come as no surprise to most of the people here. When looking at the kind of questions you've raised—for example, the arbiter—I would say that, in view of some of the past experience we have had, there is a role for randomization here. Perhaps a problem such as the arbiter problem is, in a certain deterministic sense, not solvable, and you can get results or statements about the situation that are as bad as you wish. But, if you adopt the point of view that you want to be right most of the time, and that you flip coins in order to make decisions in those situations, you might get better results. For example, Alice or Bob, instead of vacilating there and saying "Is it up? Is it down?" and so on, flips a coin. You can then try to arrange an algorithm which results in you're being right, or the system working in the intended way, most of the time.

I wouldn't be surprised if, when you go to that machine which is so badly put together, shown in Figure 11, that it will turn out in the final analysis—when we understand more about it—that there is some random behavior there that gives it the stability we perceive.

If you consider some of the other problems in this area, such as the mutual exclusion problem, it turns out that the randomizing solutions in the same kind of model that the Lynch-Fischer results were considering in fact give better stability and better behavior. That's something to be considered—especially in the context not of the three or four processor problem that you mentioned, but of much larger conglomerates of processes and processors.

Answer: I don't want to imply that any problem that I haven't mentioned isn't worth mentioning. The problems I've discussed are the ones that came to mind when I sat down and decided what I was going to talk about, and was limited to areas that I felt competent to talk about. Yes, the area of probabilistic algorithms is certainly a very fruitful one to consider.

I think Mike is wrong about the arbiter. It appears that the introduction of randomness does not help you in getting around the arbiter problem. In some sense you, can always get probabilistic arbiter solutions by simply waiting a fixed amount of time and then proceeding—that is, building your system assuming that everything will work in a bounded length of time. This is the way we now build asynchronous computers, so, in that sense, you do have a probabilistic system. But I don't think you can do any better by introducing randomness.

Albert Meyer: You can solve the problem in a random sense. That is, the expected time should be bounded, not the worst-case time. I think that's what Michael is talking about, and that would be the probabilistic solution.

Michael Rabin: Yes.

Answer: Yes, but you don't have to introduce randomness. If you tell an engineer about the problem and he's never heard of it before, he'll say: "Oh yes; if she can't make up her mind, she tosses a coin." But this just pushes the problem back one level, to how she decides whether to toss a coin or not. You can't get away from that problem. All of our asynchronous computers are probabilistic computing devices in the sense that we know they cannot possibly work 100% of the time; good engineers design their computers in such a way that the probability of an arbiter synchronization failure is acceptably low. If they're not good engineers and they don't do that, then the probability of that kind of failure gets to be unacceptably high. There have been systems which have crashed regularly with arbiter synchronization problems, until people figured out what was going on.

Question: For the last two months I've been working in industrial organizations on data communications. I believe that it's more fruitful to use the term "model" as an adjective rather than a noun. One thing I've noticed is that you have given examples of the kinds of things that you're talking about instead of giving formal definitions. We've done experiments on defining protocols by example. I wonder if you have some feeling for the ways that human beings tend to communicate by examples rather than in a formalism

Answer: In the area of protocols, as in all areas, human beings do communicate by example. I like to write papers having examples more than ones having theorems. Unfortunately, human beings very often draw the wrong conclusions from examples. The point that you're trying to make with the example is not the point that people gather from it. My favorite example of that—or my most unfortunate example—being in my *Time, Clocks* ... paper, where what I regarded as simply an example illustrating the technique was regarded by most people as the main point of the paper.

The problem with protocols is that whenever you look at examples, you're always looking at the common, obvious cases. What you need formalism for is to discover those uncommon things that you didn't think of. In the specification of protocols, I don't see any escape from formalism—that is, formalism in the sense of something you can look at almost syntactically and decide whether you're being rigorous. But again, that's an application area; it's not an area where you're trying to figure out what the fundamental theory is. Even though I can describe these algorithms and talk about them by drawing pictures, I will not believe a concurrent algorithm unless I have a very rigorous assertional correctness proof of it. People make errors.

Bharat Bhargava: I just want to know why you don't think that distribution makes some of the problems different or difficult compared to the other ones. At the lower level they all are the same, but at some level I think that the distributed problems are different.

Answer: When something is distributed, it changes. Building a file system is a different problem if it's to be built on a single computer than if it's distributed over two computers. In the same way, writing a numerical algorithm is a different problem if your division is three times as slow as your multiplication or ten times as slow as your multiplication. These are things that affect the efficiency of the implementation. Whether you have an 8-bit or a 32-bit machine is somehow not too relevant for computability theory, although it is certainly relevant in terms of implementation.

I'm not saying that the problem of implementing distributed systems is the same as the problem of implementing nondistributed systems. Nor am I saying that the specific verification techniques for a particular language construct for distributed programming work exactly the same as the techniques for shared-memory programming. What I'm saying is that the basic theory—the basic method of verification—is the same. It makes things confusing if you try to say that you're working on some wonderful new method. The world doesn't need a proliferation of new methods for doing this, that and the other thing when the principles already exist for unifying them and describing how they're all different applications of the same basic ideas.

Bharat Bhargava: So what you're saying is that there are no new principles involved in distributed computing?

Answer: I have yet to see anything that I would regard as a fundamental principle that ...

Interruption: Distributed clocks wasn't fundamental?

Answer: I'd say that the fact that suddenly, in the past couple of years, distributed processing is now a morally good term hasn't changed anything. Those same problems have been around for the past fifteen years; it's just that we're now talking about them in terms of distributed systems. The problem of clock synchronization is not basically different whether you're talking about synchronization by exchanging clock values over messages, or by having several processes that communicate use shared variables to indirectly read the clocks. It's the same fundamental principle. The fact that you call one distributed and you call one nondistributed is not relevant at a fundamental level, although it has implications for practical implementations. But in terms of a fundamental theory of clock synchronization—no, there is no difference between what's now called distributed processing and what used to be called multiprocessing.

Question: What you said is: that didn't used to be an important problem and now it is. This suggests that, if anything, distributed systems have created some problems that we didn't used to think about, and that's what people believe to be the new challenge.

Answer: I could be arrogant and say that distributed systems have now gotten other people thinking about some of the problems I've been thinking about for a long time. However, that's not as accurate as I'd like you to think it is.

Yes, distributed systems give you different implementation constraints, so certain problems are now considered more important than they once were. Other problems are considered less important. Now that memory is cheap, clever methods of trying to squeeze data into small amounts of storage are no longer interesting. Now that processes are communicating over a smaller-bandwidth channels than they used to when they were all hooked up to the same memory, different sorts of algorithms are interesting. But I don't think this has too much to say about a fundamental theory, and certainly not about how you should reason about those algorithms.

Question: CSP is a higher-level language construct and Ethernet is a low-level implementation concept, so what do you mean by saying that CSP isn't particularly good at describing Ethernets?

Answer: Certainly CSP isn't particularly good for describing the low level—that is, for describing the Ethernet. I am very suspicious of the idea that we use the same high-level constructs regardless of what the lower-level implementation is. Anybody who has done any real programming knows that this wonderful top-down approach is not what anybody ever does in practice. In practice, you have a very clear idea of what's going to be done at the bottom. The simple reason for this is that if you start at the top and keep working outward, and each module is expanded at each level, then at the end you have an exponential number of different modules. People don't like to write a large number of different modules; they want a small number of modules at the bottom. So, this successive refinement process has to somehow lead not to an exponential number of different modules, but to an exponential number of calls on a small number of actual modules. The only way that happens is if you have a very good idea of what's going on down at the bottom while you are doing your nice refinement at the top. So, in that sense, I don't think you ever design a system without worrying about what's going on down there at the bottom. It may very well be that even though you're building your system on an Ethernet, CSP will be an appropriate language. But what's going on down at the lower level is certainly not irrelevant.