Oral History of Leslie Lamport

Interviewed by:
Roy Levin

Recorded August 12, 2016
Mountain View, CA

CHM Reference number: X7884.2017

Levin: My name is Roy Levin. Today is August 12th, 2016, and I'm at the Computer History Museum in Mountain View, California, where I'll be interviewing Leslie Lamport for the AMC's Turing Award Winners project. Morning, Leslie.

Lamport: Good morning.

Levin: I want to start with the earliest time that you can remember being aware of computing.

Lamport: Aware of computing. Probably in my junior year in high school. I think I came across a book that discussed Boolean algebra, and must've discussed computer circuitry a little bit, because I got interested in trying to build a computer. Didn't get very far, but I think it was my junior year in high school, possibly senior year.

Levin: So that would've been what year?

Lamport: '55, '56.

Levin: And you were not so much aware of the state of computing devices at that time, rather more the mathematics behind computing circuitry.

Lamport: Well, I mean, I must've been aware of the existence of computers. I seemed to recall they existed in the movies. Wasn't it "The Day the Earth Stood Still"?

Levin: Yes.

Lamport: There was a computer. Or am I just making that up?

Levin: The robot. <laughs> How the robot worked was another matter.

Lamport: Yes.

Levin: I think-- yes. First time computers were-- sort of burst upon the public visibility might've been the election in which they were used for the first time to make predictions. Was that 1960?

Lamport: I don't know. There was a period of time probably when I was an undergraduate and going through large amount of graduate school where I was unaware of television, so…

Levin: So let's, in fact, talk about what happened after you became aware of these things in high school. Did you explore computing through high school and into your undergraduate time?

Lamport: Well, I tried to build a little computer, and this was while I was in high school. And I visited the IBM headquarters in New York, and they were very helpful for this, to this little kid who wanted to build a computer and they gave me a number of vacuum tubes. In those days, they would replace them on regular intervals, and so they were still good. And so I think I got to the point with a friend of building a 4-bit counter. <laughs>And-- But that impressed the interviewers when I applied to MIT. <laughs> So that had a good effect. But my first serious exposure to computers came the summer after I graduated from high school. Through some kind of city program that I forget, I wound up having a summer job at Con Edison, the New York electric utility. And-- that's New York City electrical utility. <laughs> -- and I was assigned to do some very boring stuff. But I learned that there was a computer there and I went up to the computer center and managed to get myself transferred to a job running tapes on their computer system — for the younger generation, tapes used to be a storage medium for computers — and they would be read on these large tape drives and they would also serve as input and output, and so you'd have to mount the input tape for a program, run the program, and then remove the output tape and give it to whoever wanted the results. So that was my job, mounting tapes, probably starting the programs and stuff like that. But in my spare time, and in the computer's spare time, I learned to program it. And the first program I can remember writing computed "e" to something like 125 decimal digits. That number being that the computer I was running on, which was an IBM 705, had an accumulator of 125 or so digits. And so I was able to do that basically without having to do multi-word<laughs> arithmetic.

Levin: So I guess it was natural that your first program involved an interesting mathematical entity, since your interest in mathematics from-- I gather from your resume -- goes back even further than your interest in computing.

Lamport: Yes. I guess in some sense to the extent that an elementary school student could be a <laughs> mathematician, I was a mathematician throughout my school years, even though the idea of actually being a mathematician as a profession never occurred to me until around my senior year in high school. But even so, it seemed like not a practical way of making a living, and so I started out being a physics major. And maybe my first or second year I switched to math. Although I'm not sure if the reason was that I really thought that mathematics was a career move, but at MIT in those days, mathematics was the only major that didn't require you to write an undergraduate thesis.

Levin: <laughs> Practical. As always. So you actually were thinking at some level about careers coming out of high school and going into your undergraduate days, it sounds like?

Lamport: Not in a very concrete sense. In terms of, "Oh. I think if I become a physicist I will be able to get a job doing thus and such," although it was clear that becoming a mathematician, the only thing I knew that mathematicians did was teach math. So I suppose at the time I switched to a math major in college my career goal would've been to be a professor of mathematics. I should mention that throughout my studies, I had part-time and summer job as programming. My undergraduate years at Con Edison. I went back to them for summers, this time as a programmer. And continued. And also worked part-time in the computer lab at MIT, and that was also running tapes. But as an undergraduate, I also remember doing a, working as a programmer, for someone in the business school.

Levin: So programming and math and physics all were interwoven during your undergraduate days, but you thought of them as sort of separate activities in some way.

Lamport: Well, they were interwoven temporally, but they were completely different worlds as far as I was concerned.

Levin: Mm-hm. And when did you first notice or become aware that there might be some — let's call it science — related to computing as distinct from it just being a technology for calculation or something?

Lamport: That's a difficult question. I really can't answer that. There was some point in my career at which I suddenly realized that I was a computer scientist. But I can't say when — it probably happened by before I went to SRI, which was in 1977. The first thing I published was a letter to Communications of the ACM commenting on an algorithm that had been published. It was an algorithm for implementing a hash table. And I discovered some very slight improvement one can make. Then thinking about the problem, I realized that there was a simpler way to do the hashing and started collecting data on efficiency, or something or other. And when I was in the midst of doing that, someone published a paper in CACM with that method. And for quite a while that was the standard method for implementing hash tables. But I had discovered I think three variants or techniques for implementing the idea that weren't mentioned in that paper, and I wrote another, a paper just describing those three methods, and it was returned by the editor saying it was just not interesting enough to be worth publishing. In later years, two of those methods, each was published in a separate paper <laughs>in the CACM, but by a different editor. Just an amusing little historical footnote. <laughs>

Levin: Well, you began to learn pretty early on how the vagaries of editing and review in publication work. That's a topic actually I want to come back to later. But I want to continue now with a more chronological view of things. So while you were at-- you mentioned SRI, but before you went to SRI you were at Compass for some years. How did you come to be working there?

Lamport: When I went back to graduate school to finish my degree, I needed to get a job to support myself. And somebody, actually, a professor at Brandeis, where I was a graduate student, recommended I check with Compass, which, official name was Massachusetts Computer Associates.

Levin: Yes.

Lamport: And they gave me a job<laughs> that, well, first was supporting me through the rest of my graduate studies. And when I got my degree, I was all set to take a teaching job at the University of Colorado in Colorado Springs. And Compass said, "Wait. Would you like to work, keep working for us in the new California office we are going to set up?" And I thought about it and decided, "Yeah, that sounded good." And so by the time I had actually gotten around to being ready to go to California, they said, "Whoops. The California office has been delayed, but that's okay. You can go out to California and work for us." Which I did. And I worked for them for another five years, I think. And they never did build, <laughs>-- open, a California office, but I kept working for them from California and would visit the home office outside of Boston for about a month once a year.

Levin: So that's very interesting: an early example of telework, which is, of course, these days a more common notion. What kind of work were you doing? Maybe it would be useful to talk a little bit about the kinds of projects that Compass did and what the business was.

Lamport: Well, Compass's main business was building FORTRAN compilers. But they also had random contracts with the Defense Department for doing different things. And with some companies. One of the things I did was design a file system for a computer that was being built by Foxboro, and Compass had the contract for doing software for that new computer — I don't remember exactly what. That was, I suppose-- would've been the first concurrent program I actually worked on, if the idea of concurrent programs was known. <laughs> B, but instead, it was-- actually worked, by using interrupts.

Levin: So I'm intrigued by the remote working arrangement that, in those days, it seems to me, would've been quite a difficult way to develop software. Can you say more about how you actually did that?

Lamport: Oh, I'm sorry. The-- that Foxboro project, was while I was still living in Massachusetts.

Levin: I see.

Lamport: At the time[1]., I had a friend who had some land in New Mexico that we would spend-- take a week or two vacation, and stay with them. And what I proposed to Compass was that I go spend — I forget whether it was one or two months there — working on this problem. And they said, "Okay." I did that. And while I was there I wrote a tome, <laughs> a-- I guess you'd call it a paper. Although it was handwritten. Maybe about this thick <laughs> in sheets, and presented it to them. And this I think blew their minds: the notion of using linear algebra And I learned later from observation that this tome that I delivered, it was like Moses delivering the stone tablets. <laughs> It was practically a sacred text that people studied and they did use it to build the compiler. But what it did serve is to reassure the people at Compass that I could go off by myself without supervision and actually do something useful. So they sent me off to California, and I don't remember if I actually-- what my actual assignments were, if I even had any, but I was being paid on some government contract. And it was I think about six months after I moved to California that I developed the bakery algorithm. And I think from that point it became clear to Compass that I was a researcher that they were going to support. And my memory is that I was left pretty much to myself to figure out what I was going to do and they would find contracts to support me.

Levin: So the bakery algorithm is obviously a very important milestone in your career, and I want to talk about it quite a bit later. But for the moment, I want to pursue this notion that Compass was — even though they didn't really formally have a research organization, certainly not in California — they were in effect supporting you as a researcher; is that right?

Lamport: To the best of my memory, yes. <laughs>

Levin: Uh-huh. Okay. So it seems like it was a fairly open-minded research environment, if you want to call it that. At least as far as your interests were concerned.

Lamport: Well, I think from Compass's point of view, they were happy to work for anyone who would pay them, and they figured out that the Defense Department would pay them to have me do research, and so that's what they did. <laughs> It

Levin: So the bakery algorithm, let's-- you invented that — this is we're talking early '70s now, right?

Lamport: Yes.

Levin: And it got published, if I remember, in '74; is that correct? Something like that?

[1] [At the time Leslie Lamport was working on the problem of compiling sequential FORTRAN for parallel computers like ILLIAC IV. The solution he came up with involved integer linear algebra].

Lamport: That sounds about right.

Levin: Yes. So they were also, Compass was also, comfortable with the notion of independent publication in those times. That you could go off and do this publication presumably on your own. Is that okay?

Lamport: Oh, sure. This notion of intellectual property didn't really exist as a practical sense in the computer industry, at least in terms of any kind of research in those days. I think that it was probably even before the days of software patents. I'm not sure of that. Certainly it would've been before the days of patenting an algorithm. And so, no: there was no notion that anything would be not published. And, of course, anything was published was effectively serving as an advertisement for Compass, so they were quite happy with that.

Levin: In fact, wasn't it, in those times, that algorithms more or less routinely got published in the Communications of the ACM as sort of people throwing out, "Here's my algorithm."

Lamport: There was an algorithm section in CACM. That's not where I published, because those algorithms, they weren't algorithms. It was probably somewhere between algorithms and code, because you had to submit a working program. And how could I submit a working program on the bakery algorithm when multiprocessor computers essentially didn't exist?

Levin: So let's talk a little bit more about the bakery algorithm at this point. Can you tell us a little bit about how you came to be interested in the problem?

Lamport: Oh. I can tell you exactly. In 1972, I became a member of the CACM. And I think one of the first-- no, of the ACM, <laughs> and one of the first CACM issues that I received contained a mutual exclusion algorithm. And I looked at that and it seemed to me that that was awfully complicated: there should be a simpler solution. And I decided, "Oh. Here's a very simple solution for two processors." I wrote it up. I sent it to ACM. And they-- editor sent it back saying, "Here is the bug in your algorithm." <laughs> That taught me something.

<laughter>

Lamport: It taught me that concurrency was a difficult problem <laughs> and that it was essential to have proofs of correctness of any algorithm that I wrote. And it-- well, of course, it got me mad at myself for being such an idiot, and determined to solve the problem. And attempting to solve it, I came up with the bakery algorithm. And, of course, I wrote a careful proof — what for those days was a careful proof. And in the course of writing the proof, I discovered that the bakery algorithm had this amazing property that it

did not require atomicity of reads and writes. And I understood that was a really cool thing, and I'm not sure at which point it made me realize that I had really solved the mutual exclusion problem, because I had written an algorithm that didn't require any underlying mutual exclusion: something that, there was one article by Per Brinch-Hansen that said was impossible. <laughs> So it felt nice having solved an impossible problem.

<laughter>

Levin: So you wrote up the algorithm then and submitted it with a proof.

Lamport: Yes.

Levin: Do you recall how the editor of the journal reacted at that point? Was it a straightforward, "Oh, great. I'll accept it," or something more complex?

Lamport: It was straightforward. He sent it out to referees. The referees-- I don't remember any significant comments from the referees. And it was published without a problem.

Levin: So this was piece of work you were doing while you were employed at Compass. Did you share it with colleagues there?

Lamport: I did. In fact, the next time I was out at Compass, I showed it to a colleague, Tolly Holt — Anatol Holt. And I got a reaction I don't think I've ever gotten from anyone else, which told me that Tolly appreciated that algorithm more than anyone else had. Because when I showed him the algorithm, he said, "That couldn't be right." And, you know, I showed him the proof and he listened to the proof, he went over it, he said, "Well, I don't see any hole in this algorithm, but it must be wrong, and I'm going to go home and think about it." And I'm sure he went home and thought about it, and, of course, never found anything wrong with it. But I was very impressed by this visceral approach that Tolly took to the issue. Tolly, by the way, was, he was an -- I don't know if you'd call him an advocate — but a believer in Petri nets. And he was also interested in what I would guess we would now say was fundamental issues in concurrency. And every year for about three years, <laughs> I would go back to Compass and Tolly would have his new theory of concurrency, and he would give some lectures on it. And it would start off, in some completely different way, but then it would always wind up at Petri nets.

<laughter>

Lamport: And it was all very lovely — you know, carefully reasoned — but it just struck me as being not very useful, because I realized that Petri nets had nothing to tell me about the bakery algorithm. <laughs>

Levin: Was that a conversation that you had with him about the-- your study of the bakery algorithm and the fact that his methodology for talking about concurrency didn't work with it?

Lamport: No. I don't think we ever had a discussion about that. I learned some, I think, important basic things from Tolly, and I'm not sure what they were, but I think the experience of thinking about things in terms of Petri nets was in fact useful. Tolly is probably best known for-- as being one of the inventors of marked graphs, which are a particular class of Petri nets that are quite interesting — they have very nice, very simple properties, and in fact, are a useful model of a lot of real-world computation. It just-- they weren't going to be useful to me in general or as a general way of looking at concurrency.

Levin: Coming back to the bakery algorithm itself, I've read in one of your perspectives on it that you thought it was the one algorithm that you discovered rather than invented. I'm not sure I understand the distinction.

Lamport: <laughs>

Levin: Can you elaborate on that?

Lamport: I'm not sure I understand the distinction either. But there's just-- there just seemed to be something fundamental about it. And maybe the basic idea of the use of a counter, a sequence of numbers that can in fact increase forever, that you will see elsewhere in things I've done — in the "Time, Clocks" paper, for example, the use of timestamps or that kind of counter, and in the whole state machine approach, in Paxos, there's-- well, in Paxos, there is also a counter that's being used in, not quite in the same way, but it seems like there's some fundamental thing going on there and I don't know what it is exactly, but it's certainly been quite useful to me.

Levin: All those algorithms that you mentioned are things that we'll talk about some more, because I think that's been a very significant thread in your work, starting with the bakery algorithm. You've also said it was the one you were proudest of. Can you say why?

Lamport: No, I can't.

<laughter>

Levin: Well, it doesn't need to have a rational basis, I suppose. So that happened while you were in sort of independent researcher mode.

Lamport: Let me--

Levin: Yes, go ahead.

Lamport: Let me try to answer the question of why I'm proudest of it. I think in other things that I've done, I can look back and see: "this idea developed from something else." Sometimes leading back to a previous idea of mine, very often leading to something somebody else had done. But the bakery algorithm, it just seemed to come out of thin air to me. There was nothing like it that preceded it, so perhaps that's why I'm proudest of it.

Levin: Well, certainly mutual exclusion is an absolutely foundational problem in concurrency. And to have solved it as you did in the bakery algorithm, certainly is something worthy of being proud of. As part of the publication of the bakery algorithm, you've included the proof that you talked about. Was it common for papers around that time in the early '70s to include proofs with algorithms that were published?

Lamport: I don't know about algorithms in general, but mutual exclusion algorithms, which are pretty much the only concurrent algorithms that were published in those early days, approximately, but was the most significant concurrent programming problem. And yes. Dijkstra, Dijkstra's first paper, had a proof. And what's interesting, the second paper, which I believe was a note submitted to CACM, didn't have a proof. Knuth then published the third algorithm, pointing out the fact that the second one was incorrect <laughs> and I'm sure he had a proof as well. And, yeah, the mutual exclusion algorithms always contained a proof.

Levin: So for concurrency, at least, in those days, that was the norm, even if it wasn't the norm perhaps for other kinds of algorithms.

Lamport: Right.

Levin: Okay. And your work evolved from the bakery algorithm quite a bit, but was often driven by what you had learned in studying it, and in fact, the future study of that algorithm, I think you've written, was something you did for a number of years. Did that study lead you into the approaches for verifying correctness that you eventually spent quite a bit of work on?

Lamport: My interest in correctness was parallel to my interest in concurrent algorithms. I should mention, I don't want to make this-- I don't want to give the impression that I was drawn to concurrency by these fundamental problems. The fact of the matter is that they were just really cool puzzles, and when I was at Con Edison, I had someone who took-- a manager, who became sort of a mentor to me. And he would give me these problems. One problem he posed to me was: in those days, programs were on punch cards. And you would load the punch cards into the hopper, and you'd press a button on the console. And the computer would read the first card and then put that card in memory and execute that piece of memory as a program. So the first card would usually be something that had just enough of a program in it to load the rest of--, a few more cards. So you'd have a decent-sized program that you would then execute to load your program and start that program executing. Well, I remember, he posed the problem to me to write a loader that worked all on a single card. And I worked on that problem and I would present him the solution and he'd say, "Yeah, that's good. Now make your loader also have this property," and went through, you know, few iterations like that. And I was-- I just loved those puzzles. I mean, one thing I'm also proud of is a program I wrote for the IBM 705, which was: when your program crashed, you would take a memory dump and also presumably print out as much as you can of the registers to kind of find out what went wrong. And, of course, starting that debugging process meant putting a card into the-- punch card into the hopper. And so the program I had was, the problem I faced, was, well, if you do that, well, it's going to erase part of what's there, and executing the program is going to erase some of the registers that you want to, whose contents you want to print out. So how can you do that? And there was one register whose only function was as a destination for a move operation. That as you say move something, and you move a piece of data from some location to the location specified by that register. So the problem was how do you read out that register? Because it could put-- you could put something in the register. You know, you could go anywhere in memory, so you have to look through all the memory. But it could also land on top of the program that you were trying to-- that's trying to find it. So I figured out how to write a program that even if the transmittance for the move instruction put this piece of data right, anywhere inside your program, it would still work. <laughs> And you know, I still think that was a really neat solution.

Levin: What did your boss at Con Edison think?

Lamport: I don't remember if-- I must've showed it to somebody, but I don't remember the reaction.

Levin: Because I was wondering if they actually ended up using it. You could take credit for writing a debugger.

Lamport: <laughs> Oh, no. I don't think anyone cared that much about--

<laughter>

Lamport: --that register to worry about it. But the thing about concurrency is that it added a whole new dimension of complexity, and so even the simplest problem became complicated. It's very much like-- just as trying to write a program where, you know, something could transmit a value into the middle of your program, where in concurrency you're trying to write a-- get a process to be able to do something even though another process could be doing anything else <laughs> that that process might do. So it posed that same kind of difficult puzzle. And that's what attracted me to concurrency more than a desire for fundamental theory. Once I started working on concurrency, then I became interested in what I might call fundamental problems in concurrency. But initially it was just fun puzzles.

Levin: Very interesting. That's great. So let's continue the sort of chronological thread a little bit. You worked at Compass until '77; is that right?

Lamport: Yes.

Levin: When you moved to SRI?

Lamport: Right.

Levin: How did that come about?

Lamport: Oh, for some reason that I never understood, the people at Compass said that it was a nuisance having me in California that-- they said I had to move back to Massachusetts and I didn't want to move back to Massachusetts, so I found a job in the Bay area.

Levin: Mm-hm. And how did you happen on SRI? It was…

Lamport: I knew people at SRI. I think SRI and Xerox PARC were the only places I knew that did computer science research. And I applied both places, and PARC didn't make me an offer, and SRI did.

Levin: And what was SRI's business around that time? What kinds of things were going on there?

Lamport: Well, SRI was very much like Compass in the sense that if you find somebody to pay for it, <laughs> you can do it. But the major project that-- the obvious project for me to work on — and I suppose they had it in mind when they hired me — was the SIFT project, which was to build a fault-tolerant computer system to fly airplanes, essentially. It was a NASA contract. And that was the work that-- the contract that led to the Byzantine Generals' work. So that's what I started working on.

Levin: Let's talk about SIFT a little bit more. So that was already underway when you joined SRI. And it seems like in the late '70s, if we sort of think back to the state of aviation in those times, the notion of having computers flying an aircraft was pretty audacious. Do you know how it was that that contract came about?

Lamport: Well, I don't think it was such an audacious idea. And the motivation for it, as I understood it, was that was not long after the oil crisis of the early '70s, where because of an embargo — or some kind of embargo, it's not clear exactly what was permitted and what in fact wasn't — there was an oil shortage in the U.S. and lines at gas stations. And that didn't last for very long, but it made people think about trying to save gasoline. And the one way that they were able to save gasoline or jet fuel was designing airplanes — well, designing airplane you want to minimize the friction and a large source of friction are the stabilizers: the horizontal and vertical stabilizers that control the plane and keep it flying straight. And they realized they could cut down friction and save fuel by reducing the sizes of those control surfaces. But doing so would make the plane aerodynamically unstable and un-flyable by a human being. And so only a computer would be able to fly it, because you had to basically make corrections to the control surface on every few milliseconds. And if you didn't, you could get vibrations and the wings would fall off. So I think another motivation was-- another thing that reducing the size of the control surfaces did, was make the plane really maneuverable. So you could build fighter jets that could really maneuver or do tight turns and all that stuff, but if you could get computers to fly them. So I think those were the, NASA's, two motivations.

Levin: Mm-hm. And the project was underway, as you've said, when you joined. What was your role in that project when you joined it?

Lamport: Well, my general role was member of the team, engaging in all the conversations and lots of decisions and stuff. But perhaps the question you're asking is which of the results that are still known, under the aegis of the Byzantine Generals' work, did I contribute to and which were there when I arrived. And the idea of what was there was the notion of the problem, namely achieving consensus in the face of completely unreliable processors. And the solution that did not employ digital signatures was known. The solution for—and also the impossibility, the result: the result that you couldn't do it with three processors, you needed four. And the algorithm for four processors I believe was invented by Rob Shostak before I arrived. And the generalization to 3n+1 processors was done by Marshall Pease. And that was an amazing piece of work of Pease. I don't know how he figured out that algorithm. Nobody could understand it. I could read the proof and follow the proof and was convinced that it was correct, but I still couldn't say that I understood it. And what I contributed to the original paper was the algorithm with digital signatures, which was actually something that I had done before I got to Compass. Before I got to Compass, I wrote a paper, and I think it was published in '77 — and nobody noticed it — <laughs> which actually generalized the "Time, Clocks" paper to the problem of implementing a state machine in the presence of arbitrary failures. And since in those days it wasn't clear what you'd consider as a failure, I just considered arbitrary failures, that is where a failed process-- these days, most work on failures assumes that processes failed by crashing. But it wasn't clearly the right model when-- or the useful

model, in those days, so I had an algorithm that would tolerate arbitrary failures, including processors that acted maliciously. And I didn't discover the impossibility result because it doesn't apply if you have digital signatures: digital signatures you just need a majority of processors to be working correctly. The other thing that I hadn't done is to pull out the consensus problem. Now, it's clear that if you're trying to build a system that works correctly, you have to choose a sequence of operations that that should do. And so you have to form consensus on choosing each of those commands. And I hadn't pulled out the consensus problem as something to be considered separately, as sort of, as a building block. And I'm not sure why, but I think in retrospect it's because the solution to that problem, the one using digital signatures, came so quickly to me that I didn't think much about it. And the reason the digital signature algorithm came from me is in those days hardly anybody even knew about digital signatures. I happened to be a friend of Whitfield Diffie, and-- who, of course, with Marty Hellman, published in around 1976 the seminal paper in "New Directions in Cryptography" that introduced public key encryption and started the whole modern field of cryptography. At any rate, it was I think around '74 that I was having coffee at a coffeehouse with Whit and he proposed the problem to me, which-- he hadn't-- he realized it was an important problem and he didn't know how to solve it. And I said, "Oh, that doesn't sound very difficult." And I thought a minute and literally on a napkin I wrote out a solution involving one-way functions. And that was the first solution: Whit and Marty published it in their paper, crediting me, of course. So I was one of the few people in the world who knew about the digital signature problem and knew that it was possible to create digital signatures. Although in the context of fault tolerance, creating digital signatures should be trivial, because you're only worried about a signature being forged as a result of a random failure, not as a result of a malicious processor, so it should be fairly easy to devise some algorithm that would guarantee that the probability of a forged signature is infinitesimal. Although I've never had occasion to really think very hard-- long and hard about that problem, and I don't think anybody else has either, so it's in some sense an unsolved problem of creating an algorithm that you can prove in the absence of a malicious failure has a suitably low probability of being broken by a random failure. At any rate, so that was the situation when I got to SRI: that original paper I contributed the digital signature algorithm. And perhaps the other important thing that I did was that I got them to write the paper, because I realized that this was an important result, an important problem, and publication, especially outside of academia, didn't seem to have as high <laughs> a priority. As a member of Xerox PARC, you may remember that it was known-- there's a lot of great research being done, but not very much publication and it was known as the black hole of computer science. Anyway, I sat down and I wrote a paper on it and I remember Rob Shostak not liking <laughs> the way I wrote the paper at all. And so he sat down and rewrote it and I imagine that something would've gotten published eventually, but I think it certainly got published sooner because I spurred them into action. And in the second paper, the only really new result in there was an understandable version of the general algorithm without digital signatures — a recursive version — and that was, I think, that was one that I discovered. But the real contribution of the second paper was introducing the name "Byzantine Generals". <laughs>

Levin: Which is what it's, of course, become known as ever since.

Lamport: Mm-hm.

Levin: Just to wrap up on SIFT…

Lamport: By the way, you should ask me about the origin of the name at some point.

Levin: I will.

Lamport: Okay.

Levin: Yeah.

Lamport: Yeah.

Levin: Just to wrap up on SIFT, how did that project conclude? Did the system get built? Did it get deployed?

Lamport: A prototype system was built. It was, I think it was run, on a simulator. I'm not sure if it ever flew in an airplane. There was a bit of controversy at the end of the project. The project was to include a complete verification of the code. And there was some controversy about that. And the short story is that through carelessness, the paper that was published about it did not make it perfectly clear what had been verified mechanically — what had been mechanically proved, and what hadn't. I think in the particular case, that the code that had been verified didn't include some optimization that was later made. So the final system included code that had not been mechanically verified. And there was this general sense in the criticism of it that the-- this was this multi-million-dollar project and, you know, they lied about, you know, the work at that time and they lied about what they had did, what they had done. And the fact of the matter was that this project was used somewhat as a cash cow at SRI in our group. And the actual writing — the actual proof of the code — was an onerous task that nobody wanted to do. And I managed to get off the project before <laughs> it became my turn to do that.

Levin: <laughs>

Lamport: And finally, Rob Shostak-- no, hold. No. It was Michael Melliar-Smith and Rob Shostak, in a really Herculean effort, as the deadline was approaching, wrote the proof and ran it through the prover. And it was really a, I would say, a baling wire and chewing gum <laughs> type of project as far as this proof was concerned rather than this, you know, grand effort. And I think that although the people were careless in what they published — this was not a serious offense, and I think the project was quite successful.

Levin: And do you know if it had any direct impact on the way avionics systems were subsequently designed?

Lamport: Well, I don't know how avionics systems are designed. The Byzantine Generals work certainly did. I was actually curious about that, and sometime maybe 10, maybe 15, years afterwards or so, I happened to be contacted by someone who had been at Boeing at the time, and asked them if they knew about the Byzantine Generals' work. And he said, "Yes, indeed." He said he had been the one who first read our paper and he said, "Oh, shit. We need four."

<laughter>

Lamport: Though he did say that this was in the commercial side, and he did say that Boeing had just bought McDonnell Douglas, I believe, who were doing the military aviation side. And that he-- from what he said, at that time, the people at McDonnell Douglas were somewhat clueless about these things. I don't know what goes on these days at Boeing. I have no idea what goes on at Airbus. I happened to visit Airbus and was told that they use a primary/backup system rather than the SIFT-type design, which was I guess what you'd call peer-to-peer these days. And, basically the programmers-- that was the job of the hardware people, that was done in hardware. And you can build a primary/backup system correctly taking, you know, handling Byzantine faults. On the other hand, I've seen occasion where engineers have very happily presented solutions of provably unsolvable problems. So I don't know-- so I'm not positive that the engineers who build the Airbus know about the Byzantine Generals results, so… But Airbus is very secretive about what they do, and you or I can't find out. <laughs>

Levin: You don't let it affect your flying habits.

Lamport: One thing that I've observed is that-- well, let me give you a, one little anecdote. As you'll remember at Digital SRC Lab, where we were, there was a reliable distributed file system that was built called Echo. And that actually motivated me to discover Paxos, because I was convinced that what they were trying to do was impossible and was trying to find an impossibility proof — instead I found an algorithm. But there was no algorithm behind that code: there was just a bunch of code. And I'm sure that there was kind of failure that Paxos would handle that the Echo code didn't handle. But the Echo system ran perfectly reliably. For how many years? I don't know — five years or something like that. And it only crashed once, and the reason for that crash had nothing to do with the algorithms. It had to do with a very low-level programming detail that buffers got overfull. And so whether or not the Airbus engineers will handle all possible Byzantine faults: I think that in practice if they, they're probably good engineers, and they've probably thought about the problem enough that if there are failure modes that would cause their algorithm to fail, they are sufficiently improbable that they're not the most likely cause of an airplane crash.

Levin: So you were at SRI for seven or eight years, and then you moved from an organization that was fundamentally a contract programming or contract project organization into what I think we might call a corporate research lab at Digital Equipment Corporation. How did that move come about?

Lamport: Oh, well, the actual details is that we were in a group that had, I don't know, a dozen people or so. And at some point, they decided that we needed another level of management <laughs>, and that, you know, in addition to the lab director and assistant director, there was now-- 4 of these 12 people became, you know, some managerial title and something, and I just said that was ridiculous. <laughs> And Bob Taylor had just started the SRC Lab, and it was a natural place for me to go.

Levin: So you came into what must've been a fairly small group of researchers at that time, since the lab was just getting started.

Lamport: It didn't strike me as being small, because it was probably larger than the group as SRI where I was, so…

Levin: But there wasn't a single project you were basically being brought into as you had been at SRI?

Lamport: No. It was clear that I was being brought in to-- well, to become a member of the lab. There was much more of a sense of community at SRC than there was at SRI, which may seem strange, because SRC was a much larger place. But as someone said at SRI, that grant proposals are our most important project. <laughs> I'm sorry: Grant proposals are our most important *product*. <laughs> And so there was always this question of you were working on a project and there was no sense of interaction between projects, so in a sense the lab was more splintered than it was at SRC, especially in the beginning where there was one project, which was building the computing environment. So it was clear that my expectation was that I would be continuing the same line of research I had been doing, but also interacting with the other people at the lab, because they were building a concurrent system and so they would have concurrency problems, and so I expected to be useful.

Levin: So how would you characterize the most important product of SRC, by analogy with what SRI said? It clearly wasn't grant proposals.

Lamport: I think-- well, I'm getting out of my depth, because this is a question of why did Digital open the SRC research lab and what did Digital expect to get out of it. Of course, as you know, the kernel of the people who started SRC were the people who came from Xerox PARC, where they had just created personal computing <laughs>, and so there was very much a sense that, well, this was what the lab was going to continue doing: creating the computing of the future. And so that was the purpose of the lab.

Levin: Mm-hm. And so there was a qualitative feel to that organization that was different from SRI; is that right?

Lamport: Definitely. But that, I think, had a lot to do with Bob Taylor, the lab director and lab founder, who was just a wonderful manager. <laughs> And as you know, he was your mentor. <laughs>

Levin: Indeed. Did you consider, when you were looking to leave SRI, other places to go work? Perhaps academia?

Lamport: I did talk to Berkeley and Stanford. But I must say, I was never really serious about it. I did not see myself as an academic, so it was clear to me that SRC was where I was going to go <laughs> if they made me offer.

Levin: Say little bit more about that. What is it about either the difference between SRC and academia that attracted you to SRC more than the university?

Lamport: Well, there were two things at play. First one was personal, and one was institutional, you might say. Personally, I had never taken computer science seriously <laughs> as an academic discipline. And in fact, when I started out, it wasn't. I don't think there was enough stuff you could call computer science, and certainly not in my field, that merited being taught <laughs> at a university. And this may be just my own background which, since I had studied math and physics, and perhaps if I had gone to an engineering-- been an engineering major at MIT, I would've had a very different perspective and I would've considered programming to be a perfectly fine academic discipline, but it never seemed to me that people ought to be going to a university to learn how to program. And by the time I began to feel that, "Well, maybe I actually could go to a university and teach academically respectable things <laughs> to people," I was too set in my ways. And I think that maybe would've been in the '90s or so. So that was one reason why universities didn't interest me. And the other thing is that I found industry to be the best source of problems, and I think in the early days of concurrency, the early days — you know, starting in the late '70s and going-- yeah, it was the late '70s and early '80s that we were talking about — a lot of academic research was just self-stimulated. <laughs> pPeople, especially theoreticians, contemplating their navel and <laughs> deciding what should happen. And I've always found that the real problems, you know, the important problems, were motivated by people in industry having things they wanted to do. SIFT was an example of that. It wasn't sponsored by an industry, but it was very much a real-world project. I mentioned Paxos, which was inspired by a real system being built at SRC. The bakery algorithm: not an industrial problem, but it was clearly important in programming. And what other ones? Oh, Disk Paxos, for example, another algorithm that happened because there was a DEC engineer in Colorado who said he wanted to build a system where instead of having to have three processors, he wanted to have three disks instead to get the redundancy, and that led to Disk Paxos. Fast Paxos — no, sorry, not Fast Paxos — fast mutual exclusion came because the people at WRL, the Western Research Lab, it was another DEC lab, but the people there were building a computer and they needed a mutual

exclusion algorithm that had certain properties. So yeah, I've just always found industry to be the best source of problems. Not all problems, but sometimes, I suppose, the most successful problems are ones where you think of the problems before industry realizes they have them. <laughs> But it's certainly my experience working with industry, and even before that, of programming, has been much more useful than most of what <laughs> I learned in school.

Levin: So since '85, was it, that you joined SRC, approximately?

Lamport: Yes.

Levin: Until the present day. You've been in research labs first at DEC and Compaq and now at Microsoft.

Lamport: Right.

Levin: And that interaction of real-world problems and research investigation has continued through that, essentially, this whole portion of your career; is that fair to say?

Lamport: Yes. It's sort of an irony and a failure that, although I look for real-world applications or real-world problems, I didn't interact with the people at SRC that much. And I think that I must have a prickly exterior<laughs> that didn't encourage people to come and-- come to me with the problems they were doing. And I wasn't active enough in going out and finding people's problems, because I always had things of my own that I was working on and it was always easier to stay in my office and keep working on those than try to go out and talk to people and extracting their problems from them. So it more often would come from somebody outside the lab who had-- knew about me or knew about my work and came with a problem.

Levin: You've had a lot of external collaboration around problems and their solutions with people in academia, even though you didn't want to be an academic yourself, so there are evidently at least some people in academia that want to attack real-world problems.

Lamport: It's… I don't feel like I've collaborated with that many people. I think certainly compared to most computer scientists, I think I have more single-author papers. I collaborated quite a bit with Fred Schneider, and I guess that's because Fred has, I think, a very similar sense of practicality that I do. I collaborated on a couple of papers with Eli Gafni. They both happened while Eli was visiting SRC, and-- well, one of them was the Disk Paxos problem, which as I said, came externally. The other one was on the mailbox problem, and that was a problem that I had had myself, or a problem that had occurred to me many years earlier and just Eli's being there was an opportunity for it. There was the collaboration with

the people at SIFT, at SRI doing SIFT. So that was in some sense the one time where I was really part of a group that I was actively collaborating on doing research.

Levin: So I'd like to shift gears now. And we've done a kind of a somewhat meandering exploration of your career chronologically, or at least part of it. But I think I'd like to start talking about some of the themes that have run through your career — and we've touched on a few of them briefly, but I would like to get into more detail on that. And the natural one to start with is probably the one that you worked on earliest, which we've talked about a bit: the bakery algorithm and concurrency, and concurrency algorithms. You've said that quite a bit of the work you did grew out of the bakery algorithm and studying it, learning from it. One of those was-- well, let me ask you this way. Probably the next famous paper that you produced was the "Time, Clocks, and the Ordering of Events in a Distributed System" paper, which is perhaps the one that's most widely cited. Did that grow out of the bakery algorithm study?

Lamport: Not directly. It had a very specific origin, namely: I received — well, this was while I was working at Compass, and from some connection, I don't remember why, but — I received a technical report written by Bob Thomas and Paul Johnson, which was about replicated databases. And they had an algorithm in there, and when I looked at the algorithm, I realized that it wasn't quite right. And the reason it wasn't quite right was that it permitted a system to do things that, in ways, that seemed to violate causality. That is things that happened before-- one thing that happened before something else would wind up influencing the system as if they had happened in the opposite order. Now, the reason I realized that has to do with my background. And from my interest in physics, I happen to have a very visceral understanding of special relativity. In particular, the four-dimensional space time view of special relativity that was developed by Minkowski in the famous 1908 paper. And I realized that the problems in distributed systems are very much analogous to what's going on in physics because — or in relativity — because in relativity there's no notion of a total ordering of events, because events will-- to different observers, will happen, appear to happen in different order. But there is a notion of causality, of ordering between two events, in which one event precedes another if, for every observer, it will appear that that event preceded the other. And the basic definition is one in which that is that if an event happens to some person-- and the other event happens to another person, then it's possible for a message or a light wave to be sent from the first person, when that event happens, and that it will arrive at the second person before that second event happens. And I realized that there's obvious analog of that in distributed systems which says that one event happens to one processor before it happens at another processor if there was a message or a chain of messages that began at the first processor after this event and reached the second processor before his event. And it's that notion of causality or "before" that was being violated in this algorithm. And I made a fairly simple modification to the algorithm to correct that. And that's the algorithm that was presented in that paper, along with the definition of the partial-ordering relation. Now, I should mention that I'm often-- one of the things that the algorithm used, and that the Johnson and Thomas algorithm used, was attaching timestamps to messages. That is, each process has its local clock and it timestamped a message with the time on that clock before sending it. And people have credited me with my inventing timestamps, which I didn't because I got them from the Johnson and Thomas paper. And what I didn't recognize at the time-- when I received that paper I just assumed that timestamps were a standard technique that people used in distributed computing. This is the first time I'd

even thought about the problem of processors communicating by sending messages. And so I didn't point out that timestamps were used in the Johnson and Thomas paper, and so I-- their paper has been forgotten and it remains as a, citation, one of the four citations in my "Time, Clocks" paper. One of the others being the Minkowski paper and another one being a previous Einstein paper.

<laughter>

Lamport: So at any rate, that's just a footnote. The other thing I realized, that I believe Johnson and Thomas didn't realize, is that this algorithm was applicable not just to any-- not just to distributed databases, but to anything. And the way to express "anything" <laughs> is a state machine. What I introduced in this paper was the notion of describing a system by a state machine. Now, a state machine is something that's really simple. You start-- it's something that's started in an initial state and it then takes steps which change its state and what it can do next can depend on incoming messages, in this case, or from the environment., or it can come from-- or from the current state of the machine determines what it can do next. Now, the ideas of state machines-- well, finite state machines are ancient, dating to the '50s or so. But they were all used as finite states, namely that the state had to be a fixed, bounded set, like it's a certain collection of registers or something. And it was very clear to me and to everybody that finite state machines were not terribly interesting as ways of talking about computation or about problems. But if you remove the restriction about them being finite, then they're completely general and you can describe what any system is supposed to do as a state machine. And what I realized and said in that paper, that any system would-- if it is a single system, in a sense-- what it's supposed to do can be described as a state machine and you can implement any state machine with this algorithm, so you can solve any problem. And as the simplest example I could think of, I used a mutual-- a distributed mutual exclusion algorithm. And since distributed computing was a very new idea, this was the first distributed mutual exclusion algorithm, but, you know, I never took that seriously <laughs> as an algorithm. And nor do I take this, the "Time, Clocks" paper's algorithm as necessarily a-- the solution to all problems, because although whether in principle, you can solve any problem this way, that is, building a system not without the-- without failures. It didn't deal with failures. It's not clear that the solution would be efficient, and there could be better solutions. And in fact, I never thought that my distributed mutual algorithm would in any sense be an efficient way of doing mutual exclusion. Well, the result of publishing the paper was that some people thought that it was about the partial ordering of the events, some people thought it was about distributed mutual exclusion, and almost nobody thought it was about state machines! And as a matter of fact, on two separate occasions, when I was discussing that paper with somebody and I said, "the really important thing is state machines," they said to me, "There's nothing about state machines in that paper." And I had to go back and look at the paper to convince myself that I wasn't going crazy <laughs> and really did mention state machines in that paper. I think over the years, the state machine concept — thanks in part to Fred Schneider talking about state machines — people have gotten the idea. The way I met Fred Schneider is he sent me a paper. I don't remember what it was about that he had written. He was still-- I think he may have been a grad student or possibly, I think, a young faculty member. And I sent him-- He was working along, somehow, along similar lines and so I sent him a pre-print of the "Time, Clocks" paper, which I think subsumed what he had been doing. But I thought, "You

know, this guy's… This guy's pretty good." <laughs> "I should stay in touch with him." And I was right. <laughs> So…

Levin: So this paper has been very widely cited. Obviously because you did a foundational thing, and whether people really understood that or not, somehow they decided it was the place to point to in their own papers. Do people continue to-- I mean, this paper was a long time ago — almost 40 years now since it appeared. Do people continue to discover the paper and learn things from it?

Lamport: Well, what amazed me is that I recently learned that there are physicists — or at least one physicist — <laughs> who takes the paper very seriously, and says it's very important in developing a theory of the physics of time, and — Now, this boggled my mind. I could understand-- I regard what I did in that paper, at least the partial ordering that I defined, as being very obvious, to me, because I understood special relativity. And so I figured, "Well, it seemed-- it seems really magical to most computer scientists because they don't know any special relativity." But of course physicists understand <laughs> special relativity and they thought it was a seminal paper. And so I— got me thinking, "Why?" And I finally figured out for the physicist — and I confirmed with one — that what my paper did-- the advance that my paper made, was that Minkowski was talking about, in some sense, messages that *could be sent* to define his relation, and I changed that to messages that *actually were sent*. And this seemed to me to be perfectly obvious but to the physicist this was a seminal idea. So I'm-- it seems that things that seem obvious <laughs> to me do not seem obvious to other people. I've never understood quite why the paper is considered so important in computer science. because the algorithm it presents is of no particular interest. It has historical significance, but I think that somehow what people get out of it is a way of thinking about distributed systems that's so natural to me that I consider it obvious, that is not obvious to them, and gets them to be thinking in a different way. This may be related to one difference between me and I think almost all researchers in concurrency —so, theoreticians — is that in most of the research that I've-- most of the papers that people write or people I talk to, they think of something like mutual exclusion, for example, either as a programming problem or as a mathematical problem. I think of it as a physics problem. Mutual exclusion problem is the problem of getting two people <laughs> not to be doing something at the same time. "At the same time" — that's physics. <laughs> And I look at things from a very physical perspective and-- which I think gets me looking at the essence of problems and not getting confused by language the way a lot of people do. I've observed something in computer scientists that I call the Whorfian syndrome. The Whorfian hypothesis --Whorf was a linguist early in the 20th Century, who-- his hypothesis is that the language we speak influences the way we talk. Well, what I call the Whorfian syndrome is the confusion of language and reality. And people tend to invent a language, and they think that that language is reality. And, for example, if something can't be expressed in the language, then it doesn't exist. Let me give you an example. I've talked about state machines. Well, you can model a program as a state machine, and if you model it mathematically, basically what you're doing is mathematical description of a state machine. And so-- a state machine has to consist of the complete state of the system. Now, if you consider a program, part of the state of that state machine describing the program is what I call the program counter: it's the thing that tells you where in the program you're executing, and which statement that you're executing next. Well, there's no way of talking about the program counter in most program-- modern programming languages, and so to people who are hung up

in programming languages, the program counter doesn't exist. And so there are cases when in order to reason about programs, particularly concurrent programs, you have to talk about the program counter. And I've talked to people who say that— well, at least one theoretician, he works in programming languages, said — "That's wrong. You simply shouldn't say it." Wrong-- well, it was the notion-- the programming language jargon for wrong is "not fully abstract." But he was saying that my method of reasoning about programs, which involved the program counter, which basically you have to use, must be wrong. And a lot of people have created methods that-- since you have to talk about the program counter, they have ways of getting around it by talking, introducing, other things that can act as proxies for the program counter. So that's just an example of what I mean by the Whorfian syndrome. And I think there are a lot of people who are-- through the years, have worked on problems that were meaningless outside of the context of the language in which <laughs> they've been talking about them. And that has not been my problem, I believe. I may suffer from-- the Whorfian hypothesis may apply to me equally well.

<laughter>

Lamport: But I don't get hung up in the language I'm talking about. I mean, look at the physical reality that underlies these things. And it's that way in concurrency problems. I think the reason why things that I've done have turned out to be significant years later is not that I have any more foresight than anybody else or no better way than anybody else of predicting what's going to happen in the future. It's the fact that if I'm dealing-- I've dealt with real problems, problems that have a real physical sense, that are independent of a particular language in which talking about them. And it's those problems have a good chance of turning out to be useful, whereas problems that are just an artifact of the language you're talking about are not going to seem very interesting when people stop using <laughs> that language.

Levin: That reminds me of another paper that you did about the same time. Not in the same thread as the "Time, Clocks" or the bakery algorithm, but about a problem that was very real at the time, the glitch.

Lamport: <laughs>

Levin:. Where there it seems a very, again, a very physical thing, enabled you to bring some insight to that that others, that eluded others. Could you talk about that a little bit?

Lamport: Sure. Let me explain the glitch, which is more formally known as the arbiter problem. Imagine that you're-- well, okay, it's described in some paper by some--, a housewife. (In those days, housewife was not politically incorrect. <laughs>) And the milk man comes to the front door and the postman comes to the back door and they both knock at about the same time. And you have to decide which one to answer. And so it's a problem of making that decision. And this is known-- philosophers have known of that problem, in the name of "Buridan's ass", where instead of the housewife having to decide whether to

go to the front door or the back door it's you have a donkey who has to decide which of two bales of hay to go to, and he's equally distant from the two. And it was realized in the early '70s that computers act like this housewife or this donkey, and have to make a choice between two different things. The way it happens in computers is that you have a clock that's inside the computer and you have some event that's external to the computer — maybe a message comes in or something — and the computer has to decide — it's doing one thing at a time — has to decide whether it should-- 'cause the clock pulse tells it to "go to the next instruction" or something, and the external interrupt says to "stop what you're doing and do something else." And it turns out that if you don't design your circuits correctly, what could happen in that case is, instead of saying either "do A or do B" — which in computer terms means either produce a 1 or a 0 — it produces a one-half. And computers are not very happy when-- computer circuitry is not very happy when they see one-halves and it causes them to do weird things and to get different parts of the computer to do different things and your computer crashes. And they discovered that-- I think it was some DEC engineer, who-- there was a conference that was held on this, or a workshop that was held on this problem — And he mea-culpa'd and said that a computer he designed would crash about every couple of weeks because of this problem. And it turns out, this problem is in a sense unsolvable. What's unsolvable is that it's impossible to-- if you're going to come out-- you can come out with a 0 or a 1, but you can't do it in a bounded length of time, and — which means the computer is screwed because it has to do it by the next clock cycle — and if you do it with a bounded length of time, you have a possibility of getting the one-half. So what computers do-- but you can build circuits that will almost always do it pretty quickly, and as a matter of fact, that the probability of not having made the decision within a certain length of time decreases exponentially with the time. So in practice, you just wait a little while and the chances of getting a one-half are negligible. So that's what computers do: they don't try to decide whether this event happened at the same time as the current clock pulse, but whether it happened before the clock pulsed three ti-- you know, <laughs> three pulses later or something like that. And so engineers who know about the problem, they design the circuits. That's not an issue. Engineers who don't know about the problem designs the circuits that will fail because of it. And I hope that these days all engineers <laughs> know about the problem. But if an engineer doesn't know about the problem, he'll be able, or she will be able, to solve it. You know, they'll give you a circuit that they say solves it, and of course, it doesn't. It just pushes the problem to a different part of the circuit. So that's the glitch or the arbiter problem. And when I wrote the bakery algorithm, someone at Compass pointed out this problem to me and said, "Well, your bakery algorithm doesn't require atomic operations, but it still requires solving this arbiter problem." And that's what got me interested in the arbiter.

Lamport: Should I go into the history of the papers and the publication and stuff, non-publication or not?

Levin: Well, I'm interested in what the impact was of the work that you did and that you published on the arbiter problem.

Lamport: Okay. What I did is I mentioned the--I told the problem, to Dick Palais, who was my de facto thesis advisor at Brandeis and has become a friend. And he realized that an argument saying, against— or the reason that people think that you should be able to solve the arbiter problem is that the argument

that you can't is based on the fact that real world is continuous. And people will argue, "Well, the real world isn't continuous, because you can actually build this operator that, arbiter, that goes zero to one. Well, it doesn't work. I can't-- we don't know how to do it in bounded time, but still it introduces discontinuity." And what Dick realized, that if you have the right definition of continuity, in fact, it is continuous. And theorems that tell you that, well, basically theorems that tell you that a system is a solution to a certain kind of differential equation, then it is going to be continuous. And we believe that the world is described by these kinds of differential equations. They are continuous. That result applies, and therefore this is a mathematical proof of the impossibility of building an arbiter that works in a bounded length of time. So we sent a-- wrote a paper on that, we sent it to some IEEE journal. Might've been Computer. No, not Computer. Might've been Transactions on Computers or something like that. At any rate, the engineers-- this was the oldest mathematics engineers — had no idea what to do about that, and so they rejected the paper. I should mention that this arbiter problem has a long history of being not recognized. And Charlie Molnar, who was a hardware engineer who worked on this problem, said he once submitted a paper on it and the paper got rejected. And one reviewer said, wrote, "I'm an expert on this subject, and if this were a real problem, I would know about it. And since I don't know about it, it can't be a real problem, so the paper should be rejected."

<laughter>

Lamport: Literally. That's the story Charlie tells. I had-- So this paper was rejected from Transactions on Computers. I think by that time the engineers knew about the arbiter problem, but they still didn't know what to do with this paper. Later-- a few years later, a similarly mathematical paper was actually published in Transactions on Computers, but the result wasn't as general as ours. I also wrote a paper called "Buridan's Principle," which basically generalized that not just to computer type of issues but to arbiter-- real-world problems. For example, if you consider, suppose someone is driving up to a railroad crossing, one that just has flashing lights, and the lights start to flash. Well, he has to make the decision of whether to stop or whether to keep going in a bounded length of time, namely before the train gets there. And that's provably impossible. So in principle, there-- it is possible, for him to-- that is, it is impossible to avoid the possibility that he will be sitting on the railroad tracks where the train arrives. And I remembered actually from my youth some train accident that happened where — I think was a school bus — that for no apparent reason, that went through a flashing light and was hit by a train in perfect visibility and nobody could explain it. And it occurred to me that maybe that could be an explanation. I have no idea whether that kind of thing — which is in principle possible — whether it has a high enough probability of being a real problem. And that, that occurs everywhere. And for example, you think of airline pilots have to make similar decisions, and they have to make them quickly. You know, and could accidents actually occur? So I wrote a paper and I wanted to get it to the general-- to the more general scientific community, so I submitted it to Scientific American. No, sorry. Not Scientific-- oh, I forget. No. I think it was Science, the publication of the AAAS. And it was rejected. There were four reviews. They literally ranged from, "This is a really important paper that should be published," to "Surely this paper is a joke." And the two others were somewhat in the mi-- less <laughs> positive or negative. One was somewhat positive and one was somewhat negative, and a 50 percent split was enough to get it rejected from Science. Then I-- usually, when I get a paper rejected, if I think it deserved to be rejected, or if there

was a good argument for rejecting it, I will not try to publish again. But I felt that this was worth doing again. So I sent it to Nature, which was sort of the British equivalent of Science. And I got back a letter from the editor who said, who read the paper, he said, he pointed out, "Well, I see this problem." And I realized it was a reasonable problem, in the sense that for someone outside the audience that I'm familiar discussing with. So I described-- kind of the answer to his thing. And then he said, "Okay." You know, and then he sent me something else, another issue, and I said, "Oh, yeah, that's a good point, but here is the way the-- what it should have said to-- with that,"and two or three exchanges like that, and I was never sure whether he was really serious or whether he just thought I was a crackpot and was humoring me, and then-- but after that exchange I then got an email from some other editor saying that the original editor had been reassigned to something else, and your paper had been assigned to another editor, to me I guess, and I have this question… and he came back with basically one of the questions he had already-- and at that point I just let it drop. Finally, somebody-- the paper was on my website, and somebody suggested I submit it to the "Foundations of Physics" journal, and I did, and that was accepted <laughs> with no problem, but, again, not the wide audience that I was hoping it would get.

Levin: So within computing it sounds like this principle is probably not as widely understood as it should be, still. Perhaps at the level of people who build circuits it's understood, we hope, but perhaps not elsewhere, and these things arise in concurrent systems all the time, right?

Lamport: The people who build computer hardware are aware of it. I don't know whether engineers who aren't principally computer hardware designers know about it. For example, back in the SIFT days I tried to explain this problem to the people who were building the computers that were going to run SIFT, because SIFT had to worry about all possible source of failures, and one possible source of failure was an arbitration failure, and I wanted to ask the engineers whether they understood this problem and what they could tell us about the probability of this happening. Well, in about 30 minutes I was not able to explain the problem to those engineers so that they would understand it and realize that it was a real problem. In those days the avionics people-- computers were just starting to be used in avionics, and the avionics people knew an awful lot about feedback control systems and such like that, knew nothing about computers or the digital domain. So do the people who are building avionics hardware these days know about the arbiter problem and know how to take it into account? I don't know. I sure hope so.

Levin: Agreed. Did any other work come out of your interest in arbitrer-free problems?

Lamport: A couple of things. First of all, I decided-- I mentioned Petri nets, and I mentioned marked graphs. Now, people in the small part of the hardware community that deal with asynchronous circuits and know-- really understand things about arbiters and bui-- try to build circuits without arbiters — know that marked graphs describe the kind of synchronization or-- the kind of synchronization described by marked graphs can be implemented without an arbiter. And it's significant because the kind of synchronization that marked graphs represent, seem to describe what I would call parallel computing, as distinct from concurrent computing. Parallel computing, in my use of the terms, means a bunch of

processors that are built deliberately to collaborate or-- or join together, deliberately to work together on a single thing. Concurrency, is where you have a bunch of processors, sort of, doing their own thing but they involve some synchronization with others, and that, in order to keep from stepping on each other's toes or bad things happening. And, parallelism, when you read about big data computation, where you're using a server farm to compute to-- to process all of the queries to Bing in the last 24 hours — what you're doing is parallel computation. And there are--there's software to do that, I've forgotten the names of the types, Map/Reduce. So for example, Map/Reduce is software that solves the kind of synchronization that is described by marked graphs. At least that's what I presume Map—Map/Reduce does. And, for a long time, I thought that marked graphs described precisely the kind of synchronization that can be done without an arbiter. And so I decided to-- to try to prove that and write a paper about it. Well, I discovered that I was wrong. That there actually is a weaker type of thing you can describe that-- I'm sorry, a more-- you can describe things that marked graphs can't describe, but that can be done without an arbiter. And, so I wrote a paper about that. And, I wrote another paper that realized that parallel computing is-- is important even though I generally stayed away from it, as not being that interesting. Just problems of-- of competition are much more difficult and interesting to solve than problems of cooperation. But, I just realized that there was an algorithm that had, sort of, in the back of my mind or— that indicated how you could have a-- an implementation of marked graph synchronization by multiple processors that are communicating by shared memory. And so, I wrote a paper about that describing that algorithm. And that's, I think, the extent of what I've done in terms of arbiter problem.

Levin: And roughly when was that work?

Lamport: I believe the work was done since I got to Microsoft, but I think it was the early part of this century.

Levin: So, a decade or so ago.

Lamport: Yeah.

Levin: Yeah. Has this whole area of arbiter-free algorithms, or implementations, or whatever, continued to receive a lot of attention as far as you know?

Lamport: I don't think it's ever received a lot of attention.

Levin: Okay.

Lamport: There's a small community of hardware builders who are-- well, almost all computers these days, or all computers you can go out and buy the store, work by having a clock. And they do something

on each clock cycle. So they'll do-- do something, wait for the gates to settle down, and go on to the next step, and that waiting is synchronized by a clock. This is a problem because your computers are so fast that propagating a clock signal from one end of-- of a chip to another takes a significant amount of computing time. But engineers work hard to solve that problem. There's another small band of dedicated hardware builders — Ivan Sutherland is-- is one of them — who believe that it's time to stop using clocks and to build asynchronous circuits. And so, you want to avoid arbitration whenever possible in-- in building asynchronous circuits. They have the advantage of potentially being much faster because if you're clocked, you have to go at the speed of the slowest component or the worst-case speed of the slowest component, whereas if you're asynchronous, you can have things done as fast as the-- the gates can actually do it. And I know I've heard Chuck Seitz, who's another member of that community who I used to interact with -- used to chat with him a bit and I'm sure learned a fair amount from him — he has described building, basically really, really fast circuits for doing signal processing with this asynchronous technique. But so far it hasn't been adopted in-- in the mainstream.

Levin: So, it sounds like there's at least a potential that you might do more work in this area in the future.

Lamport: I think, my days of working in that area are-- are done. But somebody else might take it up.

Levin: Okay.

Levin: Let's pick up the thread of your work in concurrency after that time. You continued to do work in concurrent algorithms and I think there was a paper that you did not terribly long after that. Maybe it appeared around the same time as the Time and Clocks paper that was about concurrent garbage collection that may be the first paper, I believe it was the first one I knew of, where you were working in some sense directly with Dijkstra, although I might have gotten that wrong. Can you talk about that a bit?

Lamport: My involvement that Edsger Dijkstra wrote with some of the people around him — loosely call it his disciples, the authors on the paper; I don't remember whether they were still students at that time or what — about a concurrent garbage collector, and-- he wrote it as an EWD report (the series of notes that Edsger distributed fairly widely) — and he sent me a copy or I received a copy somehow, and I looked at the algorithm and I realized that it could be simplified in a very simple way. Basically it treated the free list in a different way than it treated the rest of the data structure and since it's a concurrent garbage collector I realized that you could just treat it as just another part of the data structure, and moving something from the free list someplace else is the same as— basic problem as making any other change to the list structure. So I sent that suggestion, which to me seemed perfectly trivial, and Edsger thought that it was sufficiently novel and sufficiently interesting that he made me an author as the result of that — something I imagine he regretted doing. <laughter> Yeah, there was an interesting story about that. Edsger wrote a proof of correctness of the algorithm and it was in this prose-style, mathematical-style proof that people used and people still use to write proofs, and I got a letter from Edsger — those days one actually communicated by writing letters and sending them by post — and it was a copy of the letter he sent to the

editor withdrawing the paper that had been submitted saying someone had found an error in it. Now I found Dijkstra's proof very convincing and I decided that, oh, there must have been just some minor, insignificant error, and I had just developed the method of reasoning formally about concurrent algorithms that I published in, I think it was, in the '77 paper proving the process of multiprocess programs. So I said, okay, I would sit down and write a correctness proof and I'll discover the error that way and I'm sure it'll be a trivial error, easy to fix. So I started writing a proof using this method that I had developed and in about 15 minutes I discovered the error and it was a serious error. The algorithm was wrong. I had a hunch that it could be fixed by changing the order of two instructions, and I didn't understand wh--, I didn't know whether that would really work or have any very good idea of whether it would work or not, but I decided to give it a try and of course what I would do is use my method to write a proof, and it was hard work because writing that style of proof you have to find an inductive variant, which — I won't go into now <laughs>what that is — and that's hard and especially at that time I had no practice in doing it. I spent a weekend, and I was actually able to prove the correctness of my version of the algorithm. It turned out that Edsger had found a correction that involved changing the order of two different instructions because it was somewhat more efficient to do it that way, we used his in rewriting the paper, but there I insisted that we give a more rigorous proof, and Edsger didn't want to give the really kind of detailed proof that I believe is necessary for making sure you don't have these little errors in the algorithms, but we compromised and we had— the invariant, I believe, appeared in the paper and not written formally, but written precisely, and so we had a sketch of that proof. And a little while later, David Gries — I think that he wrote that paper by himself not with Susan Owicki — but David Gries basically wrote his version of the exact kind of proof that I had written. It was the same time I was developing my method David and his student, Susan Owicki — or I suppose since it was her thesis it was must've been Susan Owicki and David Gries — <laughs> were developing what was an equivalent method, logically equivalent, but it was more politically correct because — remember that I told you that you needed to reason about the PC in order to write these kinds of proofs, but they didn't reason about the PC: they introduced some way of writing auxiliary variables that would basically capture the difference — but there was another difference in the presentation, and in some sense they were doing it in a—well, this method of reasoning about programs was introduced by Bob Floyd in a paper calling-- Assigning Meanings to Programs. He did it in terms of writing-- using flowcharts and annotating the flowcharts. Now, Tony Hoare developed another way of doing it called Hoare logic in which it's-- in some sense it's a nicer way of doing things for sequential programs, and what Owicki and Gries did was present it in a language that made it look like they were using Hoare's method, but they weren't really using Hoare's method. They were using Floyd's method but in disguise, and as a result — and their method and mine were logically equivalent — their method became well-known and my method, it took about 10 years before people realized that what I was doing was really <laughs> the same thing that they were doing, and it really-- but people were really confused about-- by their method and in fact Edsger wrote an EWD calling my vision of the Owicki-Gries method, which he said was to explain it simply, and I thought that-- my reaction was, "God, the poor people who tried to understand it based on his description." <laughter> When you look at it the way I did it it's very obvious what's going on. You see it. It's clear that you're writing an invariant and each step you're checking the invariant, but the global invariant was what was hidden in Owicki-Gries method, and these days-- well, actually I can't say whether people are still using the Owicki-Gries method. They don't write things in terms of flowcharts. There was a paper by Ashcroft which preceded both my paper and the Owicki-Gries paper in which he talked about doing things directly using the global invariant, and I

thought that I was making an improvement on that by basically distributing the invariant as an annotation of the program, of the flowchart, and I've since realized that that's dumb: the correct way of doing it is just thinking directly in terms of state machines and essentially reasoning the same way Ed Ashcroft did, but in those days I still thought doing things in terms of programming languages or things that looked like programming languages was a good idea. I've since learned that if you're not writing a program you shouldn't be using a programming language. An algorithm isn't a program.

Levin: So that's a very interesting illustration of the tie-in between your work on algorithms, in particular concurrency algorithms, and the methods that you used that you developed for proving the algorithms correct, which is I think another major thread in your work. The two have been interwoven over, as far as I can tell, most of your career.

Lamport: Yeah, well, in the early days-- and there are lots of papers written about verifying concurrent algorithms — what I think distinguished me from most of them was that I was doing this because I actually had algorithms to verify, and I needed things that worked in practice, and I looked at a lot of these things and said, "I couldn't use that," and even for the little things that I was going to use and let alone try to scale them up to something bigger. David Gries and-- I think is-- well, he has a long background and I think the people who-- I don't know whether he started as a mathematician or an electrical engineer, I think he started as a mathematician — but people from the old days were, in the old days, in the U.S. at any rate, somehow seemed to be more tied to real computing than people were-- most people were especially in Europe back in like in the '70s, and so, you know, their method was, as I said, it's equivalent to mine and in practical terms of whether you could use it made very little difference which one you were using.

Levin: I want to go back to something that you mentioned in passing in conjunction with "Time, Clocks" paper, which was the use of counters, which became an important notion in subsequent work, and in that "Time, Clocks" paper the counters are essentially unbounded and that obviously had practical implications which you explored further in some subsequent work about what you called "registers". Could you tell us a little bit about that?

Lamport: As you mentioned-- actually it's the bakery algorithm.

Levin: Sorry, bakery algorithm, my mistake.

Lamport: <coughs> In the bakery algorithm the-- well, I sort of said it had no precedents. The precedent, what led to the name was something from my childhood where we had a bakery in the neighborhood and they had one of the little ticket servers that you took a ticket and the next number-- you waited for your number to be called (I suppose if I had grown up in a later era it would've been the deli algorithm), <laughs> and that's basically what the bakery algorithm does, except each process invents and basically

picks its own number based on numbers that other processors have, and it's possible for numbers to grow without bound if processors keep trying enter the critical section, and so I realized that the algorithm required multiple registers to store a number because you could run out of-- you could overflow a single register, a 32-bit register for example, in a fairly short time. So the reason why the fact that the registers didn't need-- in the bakery algorithm, didn't need to be atomic meant that it was easy to implement them with multiple word registers where only the reading of a single word was atomic, and so at the first glance it sounds like, oh, that solves the problem because it doesn't matter what order you do, reading and writing of the different words it would still work, except that if you weren't careful you'd like to-- well, you'd like to prove that the numbers, although not bounded, had some practical bound. For example, that the number would never be greater than the total number of times that somebody has been in its critical section, and it was not trivial to implement that, and so the paper I wrote — I think called "On Concurrent Reading and Writing" — gave algorithms for solving that problem, and a nice statement of one of the problems solved in the algorithm was: Suppose you have a clock in your computer, and the clock will count cyclically — you know, say for instance, one day or so: it might cycle from 0 to 24 hours and then back to 0 — but how do you accomplish that if your clock had to be multiple registers, and so the first question is "what does correctness mean?", and correctness means that if the clock is counting the time, the value that a read returns is the value of time at some point during the interval in which it was reading, and so it's a nice problem and if anybody hasn't seen the solution I suggest it as a nice little exercise. So it was the bakery algorithm that led to-- me to look at that class of algorithms. There's a funny story that the-- it turns out that you don't have to assume, if you have multiple words, that the reading and writing of an individual word is actually atomic. There's a weaker condition called regularity that I won't bother explaining, but it's weaker than atomicity, and it's a condition that-- for a single bit it's very easy to implement in hardware. As a matter of fact, the condition is basically trivial to satisfy for a single bit, and when I published the paper in CACM, I wanted to introduce the definition of a regular register and talk about how they could be used to implement, for example, this kind of cyclical clock, but the editor said that the idea of reading a single bit nonatomically was just too mind-boggling and that he didn't want to publish a paper with that. So I was forced to state that you had atomic bits even though that wasn't a requirement. At any rate, that then eventually led me to think about-- actually not eventually, but fairly quickly — thinking about what kind of registers do you have that are weaker than atomic register, and there's— actually it's a weaker type than— let me think, remember what the situation is. Oh, right. There are two classes of registers that are weaker than an atomic register. An atomic register is what you'd simply think of as reading and writing happening as if they were instantaneous. One is called safe and the other is called regular, and the paper "On Concurrent Reading and Writing"; it's safe registers are automatically regular, not automatically atomic, and safe registers are trivial to build. They're basically any way you might think of building a register would pretty much have to be out of multiple pieces would wind up having to be safe. "Safe" just means you get the right answer if you read it while nobody is writing it. So, I considered the problem of implementing-- well, first regular registers using safe registers, and then atomic registers using regular registers. While I was able to solve the first problem — not optimally, but at least solving it — and I believe that Gary Peterson later published a more efficient algorithm for doing that — and for going from regular registers to atomic registers I was only able to solve it for two processors, that is, a writer and a single reader. I proved the interesting result that you couldn't do it unless both processors were writing into something. By "doing it" I meant with a bounded amount of storage. It's easy to do it if you have unbounded counters — well, somewhat easy, but not in a bounded

way — and so I was never able to figure out how to do it with multiple readers, and after I published my paper on it, there were a couple of papers published. One which claimed to have an algorithm that I have no idea why the authors believed that could possibly work, <laughs> and then there were a couple of later algorithms that were really complicated and I realized that the reason I never solved the problem is that the algorithms, the solutions, were sufficiently complicated that when things got that complicated I just would give up. I don't know — sort of a combination of aesthetics and of not knowing if I had the mental power to deal with anything that complicated.

Levin: Was it your thought that this framework for different kinds of registers with varying degrees of complexity — or utility maybe I should say — was likely to lead to something else, or were you really focused on the problem that you had exposed in the bakery algorithm which was that you have these unbounded counters you have to deal with somehow?

Lamport: Well, I said, the problem with unbounded counters led me to think of what it means to build one kind of counter out of-- one register out of bigger ones, out of smaller ones, and then that led me into considering these three classes of registers, and the safe register is one that I said that it's-- I know that you can build them out of hardware, that's clear — and so that was the only register that I could say that physically I knew how to build, and again, I always regarded this as a physics problem.,

Levin: Exactly

Lamport: And actually when I started doing this stuff it was fairly early, I think around '75 or so, and I gave a couple of talks about it, but nobody seemed interested in it. So I put it away, and I thought I had solved the problem of atomic registers, and then sometime much later, I think-- yeah, it was around '84 — Jay Misra published a paper — I think it was Jay by himself, I don't think-- he published a lot of things with Mani Chandy, but I think that one was a paper by Jay himself — which started-- was heading in the direction of that work that I already did. So I decided to write it up and publish it, and when I got back to writing it up I suddenly realized that I couldn't imagine what kind of solution I had when I was talking about it in the '70s for the atomic register case because I probably was able to look at my slides and what I seemed to have been saying was utter nonsense, but anyway, so I sat down and came up with an algorithm, and fortunately Fred Schneider was the editor of that paper — as an editor he's incredible, was, I presume he doesn't edit anymore, <laughs> but he-- actually every paper he published he really read — and he went through the paper and he came to the proof correctness of the atomicity paper and he couldn't follow it. He said I don't understand something or other, and I said, oh, Fred, I'll go through it, got on the phone and went to explain it to him and it got to this point and I said, "By God, it didn't work." The proof was wrong, and in fact, the algorithm was wrong, and-- I've always been good at writing careful proofs, but it was before I had learned to write proofs the right way — hierarchically structured proofs — and so I realized the algorithm was incorrect so I went back to the drawing board and fortunately there was an easy fix to the algorithm and I wrote a much more careful proof, and that's what finally what got published. So I'm eternally grateful to Fred for saving me from the embarrassment of publishing an

incorrect algorithm. Not completely — there was one that I did publish, but what happened is that it was a-- I gave a talk at an invited lecture at one of the early PODCs, and then it was decided I should transcribe-- I guess a recording was made. It was decided that I should transcribe the recording and turn it into a-- and publish it in the next PODC proceedings, which I did, and in one of the slides there was an algorithm that was nonsense, and I didn't pay attention to it when<laughs> I was-- I don't know how it got into my slide, but it would never have gotten into a paper if I had actually been writing it down, but I just reproduced the slide and it was not a major part of the talk, but it was just a silly mistake or silly piece of nonsense that I'd written the slide without thinking about it. But Fred saved me from what would have been the only I think really serious error in anything I've published, although it's an interesting error that was actually in the footnote in the bakery algorithm. What I basically wrote, without using the terminology, but in the footnote said that basically a single bit was automatically atomic, and I discovered when I started thinking about atomic registers that that was wrong. It was nontrivial even to get a single atomic bit.

Levin: The obvious is sometimes not so obvious.

Lamport: Well, never believe that anything is obvious until you write a proof of it.

Levin: A good maxim. I want to pick up on a topic that you talked a little bit about earlier, but which became a major theme in your work on concurrency — really fault-tolerance — and it goes back at least to the SIFT work that we were talking about earlier on: Byzantine agreement — which of course wasn't called Byzantine agreement or Byzantine faults early on — became a topic that was pretty significant in your work and I'd like to--

Lamport: I suppose we should mention the story of why it did become the Byzantine Generals.

Levin: Yes, that would be a good thing to talk about.

Lamport: Edsger Dijkstra published very early on-- in one of his papers he posed a problem that's called the Dining Philosophers problem, and a lot of people have used that as an example of how to do this in some language or other — and received much more attention than I really thought it deserved. Edsger did some brilliant work and the Dining Philosophers was-- didn't seem to be terribly important to me, but it got a lot more attention than things that really deserved more attention, and I realize it's because it had a cute story about it involving philosophers eating bowls of spaghetti which required two forks and each philosopher just had one fork or something like that, and so I realized, I thought that the Byzantine-- that the problem was really important and that it would be good to have a catchy story to go with it, and Jim Gray had described--

Levin: The Byzantine agreement problem was very important.

Lamport: That was the Byzantine agreement problem.

Levin: I just wanted to clarify.

Lamport: A couple of years earlier I had heard from Jim Gray <coughs> a description of what he described as the Chinese Generals problem, and — it's basically an impossibility result that's quite important — that I realized a lot later that Jim was probably the first one to state it and invented the Chinese Generals story to go with it. So I decided the story of a bunch of generals who had to reach agreement on something and they could only send messengers and stuff like that, and I originally called it the Albanian Generals, because at that time Albania was the most communist country in the world and it was a black hole and I figured nobody in Albania is never going object to that, and fortunately, Jack Goldberg, who was my boss at SRI said "you really should think of something else because, you know, there are Albanians in the world", and so I thought and suddenly Byzantine Generals and of course with the connotation of intrigue that was the perfect name. So that's how it got named, but you weren't originally talking about the name, you were talking about the problem. So what was your question?

Levin: Well, I think because this is such a fundamental thing, the transition from earlier work in which you assumed that things didn't fail to work in which failure had to be factored in — was in fact sort of almost fundamental — it seems to me an important transition and obviously it's important to deal with failure. It becomes ever more important in the modern world, so I'd like to hear how you got into that.

Lamport: That was obvious because I had had the "Time, Clocks" paper out with its algorithm for implementing an arbitrary state machine, but it was clear that when you had dealt with communication you had to worry about messages being lost and at the very least, so you had to worry about failures, and so that led me to say how can I extend the algorithm from the "Time, Clocks" paper, and make it fault-tolerant. And I guess, when I was through it didn't look much like the algorithm from the "Time, Clocks" paper, but as I said, it was an obvious next step. And I wrote this paper-- this forgotten paper in '77 about implementing an arbitrary state machine. And, in-- in the SIFT work, it didn't talk about-- they didn't think in-- talk in terms of state machines. They had already abstracted the problem to sort of-- it's a kernel of just reaching agreement. And then to build a state machine, you just have a series of numbered agreements that-- on what the state machine should do next. But, in the context of SIFT, there was-- Oh, SIFT was a synchronous system, so the system did something every 20 milliseconds or something. So there was a natural separation of an agreement protocol, done every 20 milliseconds and ordered by time. So when you get to an asynchronous systems, you don't have that every 20 milliseconds. And so, the-- you use what's effectively a counter. And that somebody to propose the-- the next action of the state machine you say, "Well, action 37 has been done so now this is-- we're now agreeing on action 38."

Levin: So that-- that becomes the sequencing mechanism in that Byzantine agreement is still the building block.

Lamport: Yeah.

Levin: It's just that you don't have a-- a real time clock, you have a logical time clock?

Lamport: Oh well, but the other thing that's changed, in the asynchronous world-- at the same time there was this sort of shift of thinking, at least in my thinking from the synchronous to the asynchronous world, there was also a shift in failure model, in that people were worried about computers crashing, not about computers behaving erratically and doing the wrong thing. And this can still be a problem in-- in practice, but it's a very-- it's considered to be sufficiently rare that it can either be ignored, or you assume that you have some mechanism outside the system to stop the system and restart it again, if that happens. I do remember once in — I think it was in the-- either the '70's or early '80's when the Arpanet-- I forget whether it was the Arpanet still, or the Internet — went down, because some computer didn't just crash, it started doing the wrong thing. And since the algorithms were presumably tolerant of-- or people worried about making them tolerant for crash failures but not of Byzantine failures-- malicious failures like-- that are now-- that are called Byzantine failures, or even just failures causing computers to-- to behave randomly. The Arpanet was not protected against that. And I don't know what they did to get it back up.

Levin: So this-- this leads us, I think to the-- to another project which became whole series of-- of works of yours around building-- building reliable concurrent systems and that was Paxos. You mentioned a little bit earlier that this was motivated by some work at SRC where you were trying to-- to show how something couldn't work, and instead you got an algorithm out of it. Can you talk a little bit more about that, and then tell us a little bit about how that work-- work came to the attention of the world outside SRC?

Lamport: Oh sure. First of all, the-- before that I had-- there was a book published by, what's the-- it was a database book. And-- but they had an algorithm supposed to deal with fault tolerance. And I looked at the algorithm, and I said that wasn't an algorithm to deal with fault tolerance, because they assumed that-- that there was a leader. And if the-- and if the system failed, well you just get another leader. But that's a piece of magic. How do you select a leader, and select a leader so that you never have two leaders at the same time, and what happens if you have-- that's just as difficult as the problem they claimed to solve. So, I just ignored that. And— which in retrospect, was a mistake, because, I think, if I had thought about the problem and say, not about just how-- how their algorithm doesn't work — but I thought about how do I make that algor-- how do I turn what they did into an algorithm that works, I think I would have gotten Paxos a couple of years earlier. But at any rate, as I said, the people at SRC were building the Echo file system and I thought — again this was-- this was a-- a case where we're just interested in crash failures. People believe that that's the way processors crashed almost all the time. And so, you didn't have to worry about Byzantine failures which was good because nobody knew at the time how to do Byzantine-- how to solve-- handle Byzantine failures, except in the synchronous world. And the-- it's not practical to build large synchronous systems these days. Or I don't know, maybe possible these days, but it certainly wasn't-- wasn't the way things were going in the-- in the world today, really in most of the computing

because, to build synchronous systems, you have to have systems that will respond to a message in a known bounded length of time. And, the entire mechan-- way computer communication has evolved, there is no way to put a bound on how long it's going to take a processor to respond to a message. As you know, if you use the Net, and usually, the response comes back very quickly but it could take a minute to get a response. And so, we couldn't assume any kind of synchrony, so it was asynchronous system. And I thought that what they were trying to do in an asynchronous system was-- was impossible. There's-- well, I guess maybe the reason I may have thought it-- let me try to get a possible reason why I thought it was impossible when-- an interesting back story to that. When I was interviewing at PARC back in '85, I gave a talk and I-- I must have described some of the Byzantine Generals work, and I made the statement that, in order to get fault tolerance, or to solve the problem that I was trying to solve, you needed to use real time because in the absence of-- of real time, there's no way to distinguish between a process that has failed or a process that is just acting slowly. And I knew, I was-- I was trying to slip something under the rug which-- which I do in lectures because there's a fixed amount of time and sometimes I have to ignore a thing-- embarrassing things. Not because, I want to hide them, but just because I want to get through the lecture, but any rate, Butler Lampson caught me on that. And he said, "Well, you-- it's true what you say that you can't distinguish between whether something was failed or just acting slowly, but that doesn't mean that you couldn't write an algorithm that worked without really knowing whether a process had failed or not, but just does the right thing if there aren't too many failures or whatever." And of course, I-- I couldn't answer it on the spot. I went back home later and thought about the problem, and I came up with some argument that it was impossible, and unfortunately that email has been lost, and I don't remember what it was, but I-- I realized that it wasn't rigorous-- certainly what I know, this writing wasn't rigorous enough, and it just wasn't very satisfying and I never carried it any further. But I-- I just had this feeling that there was some-- some result there. And, some number of years later, and I don't remember when it was-- when it was published but there was the famous Fischer-Lynch-Patterson result, known as the FLP result, which essentially proved that what I was saying was impossible, was in fact, impossible. And the-- it was a fantastic, beautiful paper, and they-- what they said, was much simpler and much more general than anything I was envisioning. So that was one of the-- one of the most, if not the most, important papers on distributed systems ever written. But any rate-- that experience may have made me thought that what the Echo people were trying to do was impossible. Although, I think it was that the FLP paper came later, but I'm not positive about that. At any rate, so I sat down to try to prove that it couldn't be done. And, instead of coming up with the proof, I came up with an algorithm. I mean, to sort of say, well, an algrothim to do this, has to do this so as that can't work because of something. Well, actually this could work because of that something, but you do something else and that following suddenly you say, "Whoops, there's an algorithm here." <laughter> And but it's interesting the relation with the FLP result. The FLP result says that in an asynchronous system you can't build something that guarantees that-- that consistency in-- in reaching consensus. That is where you're never going to get two different processes that'll disagree on what value is chosen. But it says, you can't guarantee that a value eventually will be chosen. And there's-- people have showed, I'm-- that, this is an explanation, post facto explanation, not something that's going through my mind at the time — but it's been shown that there are random solutions to the problem which don't guarantee but guarantee with-- with high probability that you'll-- that is, as time goes on, as the algorithm advances, the probability that you won't have reached a decision gets smaller and smaller. And so, what I would say that Paxos does, is it-- it's an algorithm that guarantees consistency. And it gets termination if you're lucky. And it makes-- it

sort of gives engineering hints as to what you do, so that you have a good probability of getting-- of being lucky. And sort of reduces the problem to one of being lucky, which has engineering solutions. But even if you're not lucky, you're not going to lose consistency. And that's what the Paxos algorithm does. And that's what I came up-- up with. The problem with the paper was having been successful with the story with the Byzantine Generals. I decided to write a story for the Paxos paper, and it was about-- reason it's called the Paxos paper is it's-- the story line is, this is a paper's written by an archaeologist of this ancient civilization of Paxos and they had this parliament and this is how they built the parliament. And I had a lot of fun with it — the paper is called The Part-time Parliament. And so I give examples of— as part of the story or how it would be used is— I would have the story— and gave the names of the characters Greek names, or more precisely, their actual names transliterated into Greek. And, I had Leo Guibas' help in-- in doing that. I had to add a few non-existent symbols to the Greek language to-- <laugher> to get some-- some phonemes that weren't in-- in Greek. But, and I thought that people would be able to know-- people who do any kind of mathematics should know enough Greek letters to be able to know, what an alpha and a beta looks like. And so, I'm reasonably close to-- if not figuring out the names, at least be able to keep-- to read them in their heads, so they'll be able to understand the-- the story. But apparently that was beyond the-- the capacity of most of the readers. And so, that part confused them. And-- oh, I submitted the paper to TOCS, I believe, Transactions on Computer Systems. And the referees came back and said "Well, this is interesting. Not a very important paper, but, you know, it would be worth publishing if you got rid of that-- all of that Greek stuff." And I was just annoyed about the lack of humor of the referees, and I just let it drop. Fortunately, and-- and almost nobody understood what the paper was about except for Butler, Butler Lampson. And he understood-- he understood its significance in that this gives you a way of implementing and in a-- in a fairly practical way, any system. Or, any-- a kernel of a system. The part that really required that-- that keeps the system working as a unison rather than-- keeps the processors from going off in their own way and getting chaos. And, you can use Paxos as the kernel of any distributed system. And he went around giving lectures about it. Also, in the-- at roughly the same time, Barbara Liskov, and a student of hers, Brian Oki, were working on a distributed file system, I think. And they had buried inside of their distributed file system something they called Timestamp Replication, which essentially was the same as the Paxos algorithm. And some point I heard Butler say that, "Well, Timestamp Replication is the same as Paxos." And I looked at the paper that they had published and I saw nothing in there that looked like Paxos. So I would-- I dutifully would-- when I wrote about Paxos, would quote-- would say, quote Lam-- Butler, as saying that, "Well this algorithm is also devised by Liskov and Oki," without really believing it. And then years later, Mike Burrows said, "Oh, don't read the papers, read Brian's thesis." And I looked in Brian's thesis and there, indeed, was very clearly, the same algorithm. But I don't feel guilty about it being called Paxos, rather than Timestamp Replication, because they never published a proof. And as far as I'm concerned, and algorithm without a proof is a conjecture. I think, Barbara has come to realize that. And I should mention that, she and her student, Miguel Castro — well, it was Miguel Castro's thesis — actually solved the problem of Byzantine agreement, asynchronous Byzantine agreement or Byzantine agreement in an asynchronous system. And there they had a very careful proof of correctness. And so, they invent-- essentially have what's "byzantized" version of Paxos.

Levin: So, you mentioned that the-- the editors and the reviewers were not particularly kind--

Lamport: Oh, right.

Levin: --to the paper. What happened to it?

Lamport: Well, one thing, I thought it was a really terrible paper. And some point, when a system was being built at SRC and, Ed Lee — he was there. And I think it was Ed Lee who-- who was told by somebody that, "Oh, you should read this paper about Paxos, because it's what they need." And he read it and he had no trouble with it. So, I feel not so much that-- less guilty about it. And willing to shift — to put more of the responsibility on the reviews and the readers. But at any rate, it was clearly an important paper and by the-- it was finally published around '91, I think. And what happened is that — oh, names — Ken Birman. Ken Birman was the editor-- had become the editor of TOCS, and he was about to leave his editorship, and he said, "Gee, it's about time that Paxos paper were published." So he said, "I'll publish it, but you just update it to talk about what's been done in the meantime." And I didn't really feel like doing that. So, what I did is I got Keith Marzullo to-- to do that part for me. And, to carry on the story of being an archaeologist, it was that this manuscript was found in the back offices of-- of TOCS. And, the-- the author was away on an archaeological expedition and could not be reached. And so Keith wrote this-- I forget what it was called, some explanation or it was-- which was put in separate text as his contribution. So he did the annotations and it was published.

Levin: So I think if I-- if I got my chronology right — the work on Echo was in the late '80's that inspired you to work on Paxos. And the first shot at sub-- at submitting the Paxos paper was in the early '90's. And the paper didn't actually appear until late '90s, is that right?

Lamport: Oh sorry, sorry. It's the late '90's that it-- that the paper appeared. We had-- we need to check the date on that. I don't remember the-- the publication date of Paxos, but I remem-- I recall it being approximately 10 years after it was submitted.

Levin: Yes, that's early '90s and late '90's.

Lamport: Yeah.

Levin: But what stuck-- stuck in mind there. And so the-- the sort of core idea in-- in Paxos then turned out to have rather long legs, in terms of the work that you did-- that you did subsequently in applying the-- the idea again in different situations with different constraints that lead to different solutions. Can you talk about sort of that space of papers? You mentioned some of them earlier, at least in general. Disk Paxos, and Fast Paxos, and so on.

Lamport: Okay, the next one was, as I mentioned, Disk Paxos. And, when we were writing the paper, Eli sort of said, “Well, it’s really exactly the same as Paxos.” And I started writing the paper that tried to make them-- Disk Paxos look like just a variant of Paxos. And in fact, Butler Lampson, has written a paper called, “The ABCD’s of Paxos”, who claims that he has one germ of an algorithm from which he derives both ordinary Paxos, Disk Paxos, and Byzantine Paxos — the Liskov-Castro algorithm. But, I looked at the paper — I haven’t tried to look at in great detail — but I’m really skeptical, because my experience with the Disk Paxos is that, you looked at a high level, it seemed that way, where you try to work out the details, it didn’t work. And there was a-- just one, basic difference in the two algorithms. So they really are separate algorithms and there’s no—“ur”-algorithm that-- that they could both be derived from. Since that time there was another thing that happened that algorithm called Cheap Paxos and that happened because a Microsoft engineer — I had moved to Microsoft by that time — needed an algorithm that-- well the-- for tolerate a single failure, you need-- with ordinary Paxos, you need three processors. But, most of the time two processors are enough. And the third processor, sort of-- you can sort of think of it as a backup, but in particular, if the-- it doesn't need to keep up with the other two processors until there is a failure. In which case, it has to take over from one of the two processors. So he went-- the, I believe it's Mike Massa, was the engineer, Microsoft engineer. He had the idea for using an asymmetric protocol that most-- that most of the time just used those two processors and then they would have some either-- some weak processor, or just a third processor use-- have some other computer that’s busy doing other things, but can-- can come back in when needed. And, he had the-- a vague idea of how to do it and I came up with the actual-- the precise algorithm, wrote the proof, and that's how that paper came. Since then, I did some other work with Lidong Zhou and Dahlia Malkhi about— having to do with reconfiguration. One of the problems in fault tolerance is, say, for Paxos, you need three processors to tolerate one fault. Well, one processor dies, you need to replace it. Now, if it’s really just three processors, then you bring in a new-- a new computer and just call it by the same name as the old one and it keeps going. But sometimes, you really want it to move things from-- to different computers. So you want to do some kind of, what’s called, reconfiguration, where you change the processors that are actually running the system. And Paxos or in the state machine approach, it’s a very simple way of doing that. You let part of the machine state be the set of processors that are choosing the commands. Well, of course you have a chicken and egg problem if you do it the naïve way, ‘cause you have to know what the current state is to choose what the next command is. But how do you know what the current state is? And so, the-- so what was proposed in the original Paxos paper was, I said that the-- the Paxons did it by making the state, the time T, the-- that the processors would choose or the legislatures would choose the command at time T, be the one’s according to the state at T-minus-3. Or command number T-minus-3. Well, three was a totally arbitrary number. I figured that everybody would know that-- realize it’s a totally arbitrary number. But a lot of people kept saying, “What’s three? Why did he choose three?” And something. At any rate, there were-- so at any rate, we did a few improvements, optimizations, or-- how to do reconfiguration. And Dahlia said that, “This method of putting it into the state isn’t going to fly with engineers because they don’t understand it.” And so she-- we worked out a different procedure which was actually a variant of Cheap Paxos. And, but I don’t think much of that work has had significant impact. I think, it’s the basic Paxos algorithm that’s the one that’s still used.

Levin: So say-- say a little bit about the impact that it's had on-- obviously, it got-- it got used by people who were in your immediate vicinity, 'cause they learned about it quickly. But with the difficulty of the-- the word getting out through this-- this hard-to-read paper, and so on — how, how did Paxos become, shall we say, a standard-- a standard thing in building a large scale distributed systems?

Lamport: I really don't know what happened. I'm sure Butler had a-- a large effect in going around and telling people about it. I know, it's to the point where-- an Amazon engineer said to me, "There are two ways to build a reliable distributed system. One, you can start with Paxos, or two, you can build a system that isn't really reliable." <laughter>

Levin: So they obviously believe-- or he did. Are there other companies that you know of that have-- have adopted it?

Lamport: Oh sure. Google, their-- I think they call it the Chubby Locks or something like that uses Paxos. I think there are three implementa-- I've been told of that there are at least three implementations of Paxos in Microsoft systems. It-- it has become-- well, it has become standard. There are other algorithms that are in use. I-- that I don't know about. There's an algorithm called Raft which is very simple that the— well, the creators of Raft claim that one of its advantages over Paxos is its simplicity. I take that with a grain of salt. I heard somebody recently give a lecture on-- on Raft, and it didn't seem too simple to me. But, I think that what's going on is they're measuring simplicity by, you show it to something, and people say, whether or not they say they understand it. And to me, simplicity means, you show it to people and they can prove it's correct-- then prove it's correct. And somebody else had told me is that Raft is basically optimization of Paxos. There's plenty of room for optimizations in Paxos. Paxos is a fun-- it's a fundamental algorithm and there are lots of-- of optimizations you can make to it. There's something else called Zookeeper, but I don't know what its relation to-- to Paxos is.

Levin: And at least some of this is in the open source world these days, right?

Lamport: Yeah, I'm sure there are open source implementations of Paxos since the patent, I believe, has expired.

END OF THE INTERVIEW

