Scott Hanselman:
Hi, this is Scott Hanselman, and on behalf of the Association for Computing Machinery, this is *Bytecast*. And we're doing this as a special relationship between ByteCast and the Hanselminutes podcast that I host every week, so you'll find this episode on both of those podcasts. And if you're listening to this on *Hanselminutes*, I would encourage you to check out some of the great podcasts over at acm.bytecast@acm.org. Today, we're talking with Dr. Leslie Lamport. How are you, sir?

Dr. Leslie Lamport:
Just fine. Thank you.

Scott Hanselman:
It turns out you and I worked for the same company. I got to look you up on our internal directory. I work on C Sharp and .NET over in the developer division.

Dr. Leslie Lamport:
And I'm over in building 99 in research, except I actually work in California.

Scott Hanselman:
So I actually work remotely in Portland, Oregon. My entire time at Microsoft has been remote. Have you been remote the entire time as well?

Dr. Leslie Lamport:
Well, there used to be a Silicon Valley lab of Microsoft research, and that was closed. So now I'm at home mostly.

Scott Hanselman:
Are you finding that to work out pretty well? I'm personally finding it quite awkward, and I'm not really enjoying the video call lifestyle.

Dr. Leslie Lamport:
Well, I miss having a laboratory that I can be around and talk to people and just socialize with colleagues. Since I tended to work a lot by myself, it doesn't make that big a difference in terms of the actual work that I'm doing, but it is certainly a social change in my life.

Scott Hanselman:
One of the things that I'm focused on in my job at Microsoft is on making new developers feel welcome. As I was doing research on your work and thinking about and reflecting on my own, I realized that I've got about 30 years creating and writing software. If it's okay to say you're

coming up on somewhere around 60 years of creating and thinking about computer science problems and making software.

Dr. Leslie Lamport:
Well, I wouldn't say 60 years of thinking about computer science problems. It wasn't until, I don't know, some time in the late seventies or early eighties that I said, "Gee, I seem to be a computer scientist." I'm not sure what I was before that, although I did start out just programming in my spare time or in part-time jobs while I was in school.

Scott Hanselman:
I went and read your first paper that you wrote as part of Bronx High School in 1957 with the mathematics bulletin, so it sounds like you were a mathematician until you woke up one day and you were a computer scientist. Was that a bright line or did it just happen?

Dr. Leslie Lamport:
Well, When I got my PhD which was in 1972, I wound up with the choice between continuing what I was doing, which is what I would now call research at Massachusetts Computer Associates, or going off and teaching mathematics. And for somewhat random reasons, I decided to keep doing what I was doing with computers.

Scott Hanselman:
When I'm introducing young people who are entering school into modern computing or modern software engineering, they feel at least now in 2021 that there's a clear fork in the road that is telling them to either turn left and go into computer science or turn right and go into software engineering and the practice of software. Do you think that that is a clear fork in the road for people entering school?

Dr. Leslie Lamport:
Well, it's not that clear a fork in the sense that a lot of people in computer science, a lot of academics seem to go off and work in industry. It's less common for someone in industry to go and become an academic, but that happens as well. So it's not an irrevocable decision, but certainly it is a decision whether you're going to be working at a university or a research laboratory or as a software builder.

Scott Hanselman:
But you do think that they are different things though? I remember learning computer science and then parts of it, the parts of the theory and the compiler theory and the operating systems theory, I left behind as I started thinking about how agile groups work together and how projects and how humans and the squishy parts of computers work together. And I'm trying to understand that relationship between the kinds of work that happens at the mathematical level

and then at the computer science level, and then when the bits actually happen to do their thing in the processor.

Dr. Leslie Lamport:
Well, my view, and it's a view from a distance since I'm not very close to academia, is that the universities are making that split in the education, and that as a result, people who go off building computer systems never really use the computer science that they should, even if they learn that computer science in a classroom, because they may have learned the computer science, but not really how to apply it in practice.

Scott Hanselman:
Yeah. I wonder sometimes where that line should be and whether someone can be an effective engineer with a incomplete perspective on computer science, and sometimes the analogy I use is, "Does one need to be a mechanic to be a successful user of a service like Uber? If I'm driving a car, how much of a mechanic do I need to be? Do I need to be able to disassemble the engine? Do I need to talk to you about the theory behind the internal combustion engine? I'm just trying to drive to the store."

Dr. Leslie Lamport:
I don't think that's a very good analogy. Well, how's about this? If I'm going to be an electrical engineer, how much mathematics do I need? Do I really need this calculus stuff that I've learned in school? I think you do need it if you're an electrical engineer. No, math is a very basic part of engineering, and not being able to think mathematically, I don't think you'd make a very good engineer. On the other hand, you have software engineers who basically think that mathematics is irrelevant to what they do, but I don't think one can be a very good software engineer without the ability to think mathematically.

Scott Hanselman:
When you say mathematically though, there is a spectrum of math, and the way that it's taught in most countries is as you kind of go from algebra and geometry and trigonometry, and then up through calculus and then into advanced math. At some point, lay people stop, and that's as far as they went. And not everyone gets a PhD in math or gets even into graduate level math. When you say, "Think mathematically," what are we talking about, basic algebra proofs or Boolean algebra?

Dr. Leslie Lamport:
Let me put it this way. I use math. To say that I use math is a separation, as if "Now I'm doing something, so I'll start using my math." It's like math is part of me, and everything that I do and have done in computer science is done with having math as part of me.

Scott Hanselman:
The math is part of you.

Dr. Leslie Lamport:
Yeah. Now you see if you're talking about... But all the math that I use, by the way, almost all of it I learned in high school, the actual subject matter that I use. I have a PhD, but in terms of the actual mathematical stuff that I use, I think I learned most of it in high school. But you've probably learned it after you took something called a discrete math class or some basic math class in the university. So math, it's not the subject matter. The body of knowledge in mathematics, that is what's important. It's the ability to view the world in terms of abstractions, and that's what mathematics is all about. It is abstracting the real world into some kind of mathematics, or I wouldn't even say mathematics. We all abstract the world into some mental concepts. Not having mathematics sort of built into you as a way of creating mental concepts limits your thinking ability.

Scott Hanselman:
Okay. So seeing if I can paraphrase a little bit and understand, I hear a lot of people right now trying to teach young folks, even children as young as 10 or even younger, how to code, but one of the things that I've personally advocated for is less focusing on how to code and more focusing on how to think about systems, how to think about layers of abstraction, encapsulation, and how the world works as a system and how to debug problems where there is a layer that is hiding something from you or layers that go many layers deep. And then I tell them that the coding part will come later. The syntax of the text is really not coding.

Dr. Leslie Lamport:
Yeah. The concept that should be taught is basically what I call a state machine, namely that if you look at the scientific view of the universe, and it's especially manifest in physics, in classical physics, in non-classical physics as well, that the universe has a state that evolves with time. And the idea that you throw a ball, well, the state of the universe is changing in such a way that the position of that bull is changing with time as it goes from my hand to your hands. That's a really great way to think about computer systems, but what makes computer systems different is that we can view them as instead of a continuum of states as a sequence of discrete state changes where you actually go one step of a program, then the next step, then the next step. Having that as a fundamental concept to build your understanding of what a program is gives you that mental model that is useful in so many contexts of computing.

Scott Hanselman:
When the final executable code gets run, when everything happens and finally turns down into machine code and the software is run, it's usually an overwhelming amount of detail. There's a huge amount going on, but in the things that you've worked on, the systems you've worked on,

and the formal specification language of TLA and TLA+, it's really quite precise. Do you think that modern computer languages are too verbose and too detailed and exquisite in their expression of what's supposed to happen?

Dr. Leslie Lamport:
Well, if you're chopping down a tree, considering what a complicated thing there is, there are thousands of leaves on this tree, each one's following its own trajectory as you chop the tree and the tree sways and the vibration goes up to it. Now how do you keep track of all this enormous complexity in the world? Well, if you're a lumberjack, you have this wonderfully simple extraction. The tree is standing there. I chop it, and then the tree is on the forest floor. Bingo. One step. Well, you have to be able to view what's going on in those billions of gates inside of your computer as some higher level of abstraction. And the code, you need to be able to view it in terms of higher level abstraction. You can't keep those millions of things that are actually going on inside of your code in your head at one time, so that's what computer languages do. All computer languages starting from machine code hide all that complexity that's going on in the circuitry, and then higher level languages hide a lot of the complexity that's going on in the machine language.

Dr. Leslie Lamport:
And I think as programming languages evolve, people get I think probably better at building compilers more than building languages. Languages get more abstract and easier to use, but we're a long way from being able to write a web browser in a hundred lines of code.

Scott Hanselman:
And the web browsers that we have, we certainly don't know if they're correct. I can drop a web browser control on a form, but I'm sitting on top of thousands and thousands of layers of abstractions, all assuming that everything's being handled correctly. I think in the example of the tree, we can pretty much count on the universe and math and God and the creator to pretty much handle all of the state machines happening inside of the tree so the tree is either downed or the tree is up. But we'd be placing a lot of trust in the underlying abstraction layers, and what do you do when something 35 layers down has a problem?

Dr. Leslie Lamport:
There's no easy answer to that because as I say, our software has grown by evolution, not by intelligent design.

Scott Hanselman:
That is a very good analogy.

Dr. Leslie Lamport:

It's not an analogy. It's a statement.

Scott Hanselman:
Pardon me.

Dr. Leslie Lamport:
At every step people have done things just good enough so that they'll solve the particular task at hand. And then they get used for other things, and nobody knows exactly what they do in those cases. When you need complete reliability basically where people's lives depend on things, you pretty much have to build everything with intelligent design, which means you have to start not very much above the actual hardware level and verify by some reliable method that each piece does what it's supposed to, each layer does what it's supposed to. And that's not the way most code is written.

Scott Hanselman:
Indeed.

Dr. Leslie Lamport:
Fortunately, most programmers don't write life critical code.

Scott Hanselman:
So the TLA temporal logic of actions, logic that you developed, and TLA+ and the things that have been used against different services, is all based on math and logic and being able to express the intention and the correctness of something, but it sits on top of an imperfect thing. So you've got a near perfect concept of this based on math trying to describe what we know to be an imperfect thing, and then find its correctness properties and find the issues in the systems design. Does it assume that there's always going to be some bugs we can't find, or it finds the bugs and then we work around them? I'm trying to understand the relationship between the perfection of math and the obvious imperfection of computers in general.

Dr. Leslie Lamport:
TLA is not a magic bullet that's going to solve all your problems. What happens is that when you decide you're going to build a system, you start figuring out what it does. If they're careful, and I think any good engineer understands that you don't just sit down in front of a machine and start writing code is the first thing you do if you're going to build some system. You have to decide what it does, and you have to decide at a very high level how it's going to do that. And the way most systems are developed people think as hard as they can and try to see all the problems in their design, but they'll miss one. They'll miss something and not discover it, miss some corner case that they didn't think of. And if they're lucky, they'll catch it in testing. And then they've got

tens or hundreds of thousands of lines of code, and they've got to fix that problem in this really complicated mess of a thing.

Dr. Leslie Lamport:
What TLA does is it forces you to write that higher level conception in a completely precise manner, and being able to test it and find bugs at that level, and so when you're down at the code, you're just worried about coding errors. You're not worried about design errors. That's what TLA is about.

Scott Hanselman:
Why do you think that even now in 2021, in the 2020s, that we are still as software engineers doing a fairly ad hoc approach of things that might go wrong? We make a list of all of the different wrong states and all the different, "This could go wrong, and I'll have to catch that. And this could go wrong, and I'll have to catch that," while TLA is more of an express and specific, "This is what will go right," language.

Dr. Leslie Lamport:
Well, when I started working on fault tolerance back in the late seventies, what I learned is that the right way to do things for fault tolerance is to think about what has to go right, not what can go wrong. And I don't think I've ever written that down anywhere or any place, and so I was surprised and delighted when somebody said an engineer, it was an Amazon engineer, who said "Thing about TLA is that it makes you think about what has to go right rather than what can go wrong." It just seems to force you to think the right way in that respect.

Scott Hanselman:
Sometimes what can go wrong is unclear, it's confusing, and you get false signals from your system telling you that things have gone wrong. And in the eighties, you wrote about the Byzantine Generals Problem. What should someone know about that today?

Dr. Leslie Lamport:
I suppose the question is what should who know about it?

Scott Hanselman:
That's a great clarity there.

Dr. Leslie Lamport:
The average programmer writing his web server doesn't need to know anything about it. People who are building distributed systems or a fault tolerance systems of any kind need to be aware of it. Whether they need to know how to tolerate Byzantine faults, that's an engineering

decision, but they should certainly be aware of the possibility and need to make an engineering judgment on whether that is likely enough to occur to mean that that has to be considered.

Scott Hanselman:
I'm curious if you remember that the Two Generals' Problem was a thing that was a thought experiment and that was thought about and created in the 1800s. And the Byzantine Generals Problem, was that postulated by you and some other mathematicians? Where did the name come from because that term Byzantine not just means the empire, but it also means something that is overly intricate and sometimes it's used in the context of something being a fairly labyrinthine and Baroque solution? How did that term the Byzantine Generals Problem come up?

Dr. Leslie Lamport:
[Crosstalk 00:21:03] very simple. When we were doing the work, I realized that this was important and people should know about it. And I learned actually from Dijkstra and his dining philosophers problem that a cute story makes something popular, and so I created the story of the generals trying to come to some agreement and decided that they had to be some nationality of generals because what you might be calling the Two Generals' Problem I knew as the Chinese Generals Problem. And I decided I would just take a country that nobody would get offended by, and I chose Albania because Albania was basically a black hole in those days. It was a communist regime that even closed itself off not only from the west but also from the Soviet Union, and so I figured there wouldn't be any Albanians around to object. So I originally called it the Albanian Generals Problem. Jack Goldberg, who was my boss at the time, said that "You really shouldn't use that because there are Albanians around, and you shouldn't offend them." And so somehow then the name Byzantine came to my mind, and it was obviously the perfect name.

Scott Hanselman:
Indeed. The term Byzantine being associated with something being complicated and confusing and intricate-

Dr. Leslie Lamport:
No, not confusing or complicated as underhanded or... What's the term? I'm blocking on the word for-

Scott Hanselman:
There was a sense of secrecy in spycraft associated with it as well. So today the concept is that a component can be inconsistent and unreliable, and it can present different systems to different observers. And when you were thinking about this, which really applies today in our very, very

complicated distributed systems, were there a lot of distributed systems of sufficient complexity or was this still a largely theoretical exercise?

Dr. Leslie Lamport:
On one sense it was purely theoretical. Well, no, it's not purely theoretical. Well, let me give you the history. I developed an algorithm, my Time Clocks paper, of how to implement an arbitrary distributed system by basically having the components of the system collaborate to simulate, to execute a single state machine in which every component could issue commands. The algorithm would guarantee that everybody would execute the commands in the same order so they'd have the same state machine. And then I knew that any system could be described by a state machine. Whatever you wanted to do could be described by a state machine, which was perfectly obvious to me but apparently not obvious to many other people. So I realized that this was a general approach for building distributed systems, but there was no notion of fault tolerance in it. And so I decided that I should find an algorithm that used that same approach, implementing a state machine, even with processes that were faulty. There was no idea of, "What do faults do in processors?" So it just seemed natural to just assume that a faulty processor could do anything.

Dr. Leslie Lamport:
And I came up with an algorithm that essentially was the algorithm in the Byzantine Generals papers that use digital signatures. When I got to SRI, I found that people there were building a distributed fault tolerance system for flying airplanes, and they had realized that they needed to handle arbitrary failures if they wanted the kind of high reliability that needed for software, that the computer had to fly the plane otherwise because the pilot wasn't able to do it without the computer. They had the nice idea, and I don't know which of the people there had it, of abstracting the problem as the consensus problem, namely, instead of thinking about implementing a state machine, thinking about the problem of agreeing on what the next command of the state machine should be. The algorithm I had used in my algorithm basically did in fact implement a consensus algorithm. It implemented consensus on each command, and it solved the consensus problem before it was sort of pulled out as a concept by itself. So as you see, my thinking about arbitrary failures both predated and was inspired by practical considerations.

Scott Hanselman:
Speaking of the practical, how different in your mind are the distributed systems that we create in a cloud like Azure or AWS now versus the distributed systems of the '60s and '70s? A lot of your papers talked about interprocess communication and communication on different processes on different machines. Has a lot changed in 40 or 50 years, or is it just simply a matter of, we've just turned the dial up to 11 and it's just a matter of scale?

Dr. Leslie Lamport:
I suppose it depends on which work you're talking about, but the kinds of algorithms that I was considering for example in that first Time Clocks paper, a cloud system is not fundamentally different. It's just instead of, I was thinking in terms of a half dozen computers, people are thinking in terms of a thousand. And so you need different algorithms, but sort of the basic theoretical underpinning is the same.

Scott Hanselman:
And I'm curious as someone who's got a couple of decades of historical context, I'm always kind of impressed. I always look back and I go, "Wow, that was very prescient. That was very thoughtful of the elders whose shoulders that we stand upon, that they thought about that." Do you ever look back and go, "Huh. That is a surprise. I'm glad that that worked out so nicely," or is it different, and then you just feel constantly affirmed in "The early work was correct?"

Dr. Leslie Lamport:
It's not something I think about very much. It's not a question that I've ever asked myself at least in that form. One thing that's funny in a sense is that the malicious generals was a metaphor, if you like, saying that we're just not sure of what kind of failures can actually occur. And digital signatures that I used, I actually regarded them more as a metaphor because I believed and I think I'd still believe it, although nobody has ever done it, I believe that building the digital signatures for dealing with random faults should be much simpler than building digital signatures to handle malicious people trying to forge signatures. There was nothing difficult about building digital signatures, and in those days the computers were a lot slower and the digital signature algorithms were very, very expensive.

Dr. Leslie Lamport:
And so people regarded the digital signature of the algorithms that involved digital signatures as being impractical. And I would say, "No, no," because there must be simpler ways, engineering solutions that would guarantee that the probability of a signature being forged because of hardware failure could be made small enough that we didn't have to worry about it. And an algorithm based on digital signatures would become not very expensive, but nobody ever went that way despite the advantage of the digital signature algorithm in particular [inaudible 00:29:34] they require fewer processors. Yeah. I believe that's the case. It's been a long time since I've thought of this stuff. Now what's happened is we're dealing with failures that really are malicious because there are bad guys out there trying to mess up our systems, so what started as a metaphor of malicious generals has turned into a reality.

Scott Hanselman:
Yeah. I think that the initial internet, the ARPANET, didn't really consider truly malicious actors, did it?

Dr. Leslie Lamport:
No, and I believe that the ARPANET or early internet was crashed when some node became malicious and was giving out inconsistent routing information, bad routing information. I think what had happened is it sort of told everybody else that the best rate, the best path to everything, was through it. And so all messages got sent to that node, which ignored them.

Scott Hanselman:
As we get towards the end of our conversation here, my last question for you is, what work are you working on now that gets you excited to wake up in the morning and move forward? What makes you happy right now?

Dr. Leslie Lamport:
At the moment I've just, and this is for the past few weeks, I discovered what seems to be a new distributed mutual exclusion algorithm that, not particularly useful, but I think is really nice, but that has two properties that I think make it interesting. I thought it had two properties that made it interesting. One is I thought it was actually an implementation of the bakery algorithm, and the other is that I thought that the algorithm from my Time Clocks paper is an implementation of this distributed mutual exclusion algorithm.

Dr. Leslie Lamport:
Doing this work I've realized that the algorithm is not a refinement or an implementation of the bakery algorithm but a refinement of some more fundamental "algorithm" that the bakery algorithm is also a refinement of, and that I still believe that the algorithm from the Time Clocks paper is an implementation or refinement of this distributed mutual exclusion algorithm. So I think that's really neat because I and I think a lot of people had the feeling that those two algorithms are related in some way since the bakery algorithm in each process maintains numbers that can grow without bound. And those numbers are used to order the sequence in which processes entered their critical section, and the distributed state machine algorithm uses logical clocks, which are numbers that increase without bound and are used to order the execution of the individual processes commands. So there was that similarity, but there has been no sense of any rigorous relation between the two. And it seems nice that going on 50 years from the date of those algorithms that come across a precise relation between them I think is pretty cool.

Scott Hanselman:
That is pretty cool. Yeah, that was in 1974 when that first paper was written on the bakery algorithm. I'm curious, as I'll sneak one last question in as we exit here, how do you come up with these? Are you walking around your neighborhood? Are you sitting quietly with your fingers

steepled in front of you? Are you staring into the long distance? Do they come to you when you're in the shower?

Dr. Leslie Lamport:
Very few things just sort of pop up spontaneously into my brain. Some things may if I'm thinking about a particular problem. For example, just the other day, I came up with a generalization of Peterson's algorithm to multiple processes. There is something that is a generalization to it, but this is a different one. And that came about because I was thinking about mutual exclusion, but this didn't come out of a vacuum. I'm writing a web tutorial on [+CAL 00:34:13], which is an algorithm language that's based on TLA. I was looking for an example, and someone suggested an example, well actually the algorithm from the application from my Time Clocks paper of using that general state machine approach to implement a mutual exclusion algorithm. But I looked at it and I thought, "Maybe that would be a nice example for the tutorial." But signage could be made simpler. And when I started simplifying it and eliminating as much as I could from it, I came up with this distributed mutual exclusion algorithm.

Scott Hanselman:
It sounds like it's an organic process, and I think that the myth of the Eureka moment is just that, a myth.

Dr. Leslie Lamport:
Well, no. Archimedes, according to the story, didn't suddenly come up, "Gee, how could I weigh a piece of gold without destroying the gold?" The king allegedly supposedly said, "How can I tell whether this crown is really made of pure gold?" So that Eureka moment didn't come out of thin air. It came from Archimedes having a problem to solve. And almost everything I've done I can sort of point to something and say not necessarily what inspired the solution but what got me thinking about the problem. And then some times there's been a Eureka moment where it really took a good idea. And actually in my work, it seems that more often than not there was some particular problem, and I realized that this particular problem was a special case of a more general problem that nobody had ever thought of before and that my contribution was that more general problem.

Scott Hanselman:
Well, we very much appreciate you and your work and the things that you've offered us as both software engineers and computer scientists. Thank you so much for chatting with us today.

Dr. Leslie Lamport:
Well, thank you. It's been fun.

Scott Hanselman:

This has been a dual episode of *ACM Bytecast* and *Hanselminutes*. We'll see you again soon. And I would encourage you to check out both podcasts for great interviews with great and interesting people. Thank you very much.