EPISODE 1078

[INTRODUCTION]

[00:00:00] JM: Every software company is a distributed system, and distributed systems fail in unexpected ways. This ever-present tendency for systems to fail has led to the rise of failure testing, otherwise known as chaos engineering. Chaos engineering involves the deliberate failure of subsystems within an overall system to ensure that the system itself can be resilient to these kinds of unexpected failures.

Peter Alvaro is a distributed systems researcher who has published papers on a range of subjects, including debugging, failure testing, databases and programming languages. He works with both academia and industry. Peter joins the show to discuss his research topics and goals.

We are always looking for new show ideas. If you have a show idea, you can go to softwaredaily.com and create a quick posting about that show idea, whether it's a project that you're working on or the company that you work at. We are always looking for new stories to cover.

[SPONSOR MESSAGE]

[00:01:07] JM: Triplebyte is a platform for finding a great job faster. Triplebyte works with more than 400 tech companies, including Coinbase, Zooks, Dropbox and Facebook. Triplebyte is focused on matching high-quality engineers with great jobs. We know that the economic downturn has caused some significant upheaval for current computer science students and new grads and many students don't have the internships or the engineering roles that they'd expected, and you can learn more about these opportunities by going to triplebyte.com/sedaily to sign up for the webinar that Triplebyte is hosting May 28th at 5 PM. You can go to that webinar to find out more about how to prepare yourself for a job, prepare yourself for an interview. Triplebyte.com/sedaily, and during that webinar, Triplebyte will interview Tito Carriero, a current chief product development officer at Segment, as well as former early engineers of both Facebook and Dropbox, and you'll get to learn more about how to get in the door at these opportunities for internships and jobs. You can also go to Triplebyte to find resources on engineering assessments, mentorship and other materials. You can sign up for all these at triplebyte.com/sedaily.

Thank you to Triplebyte for being a sponsor.

[INTERVIEW]

[00:02:41] JM: Peter Alvaro, welcome to the show.

[00:02:43] PA: Thanks, it’s great to be here.

[00:02:44] JM: You are a distributed systems researcher. How did you first get interested in distributed systems?

[00:02:50] PA: I have a fairly unusual path into academia. I studied literature in my undergrad. I got a bachelor of arts in literature. So I was unemployable for a number of years, and my first tech job was on the West Coast in 1999 at an early search engine called Ask Jeeves. I was hired at the time because they wanted editors to build out their knowledge base, but eventually I transitioned into engineering roles. By the time I left about 8 years later, I was a senior software engineer. I had been struggling with issues related to managing and building and debugging distributed systems for a number of years when I went back and did my PhD.

Although I was doing my PhD in the area of databases or data management systems, I always in the back of my mind had these kind of pain points that I had had as an engineer in the industry. I had felt like –The large part in this informs a lot of my thesis work, I felt like we were using the wrong languages to program these systems. Do you know what I mean? I wanted to rethink how we might program those systems, and that was a great time to go and do a PhD and be able to do some really blue sky work.

When I left the PhD, although the PhD was in databases, my passion all along had been making distributed systems more programmable, easier to reason about, easier to debug, easier to verify and so on and so forth. When I joined the faculty at Sta. Cruz, I didn’t really have to choose, because nowadays, pretty much all data management systems are distributed. Pretty much all distributed systems manage data. I've been able to kind of blur the distinction between data management and distributed systems. I know that's a very long answer to your question, but it's sort of my personal history and forms my interest in the area.

[00:04:28] JM: Very cool. Your research has a number of different focuses today, and when I go on your website, you make reference to something called disorderly programming. What is that?

[00:04:40] PA: This is a conceit that I came up with when I was doing my PhD work at UC Berkley with Joe Hellerstein and my collection of colleagues that we called the Boom Team. This is very related to my answer to the previous question actually, because here we are, we're a group of database people. We know query languages. We know logic, and we're trying to rethink how you might program distributed systems in order to make them easier to understand.

Not surprisingly, one of the answers we came up with was, "Well, what if you kind of squinted your eyes and zoomed up to 30,000 feet and you kind of viewed that distributed system, all these computers, with all these states represented in different ways, as just kind of a big database and you blurred the distinction about how the state was managed." Well, if your distributed system is really just this database, then programming the distributed system, maybe, could look like querying a database. Could we get the kind of clean semantics that we get out of query languages for distributed systems programming? If we could, a lot of things that we consider to be hard now will become easy later.

In the early days of this sort of beginning to ask and answer these research questions, it occurred to us that one of the key benefits of query languages is the fact that they are order insensitive by nature. In general, when you issue a query, it's evaluated in a batch fashion against a batch of records that might be partitioned in a variety of different ways, but you don't care, because at the end of the day, you're executing select, project, join, max, min, account. These things all have the property that they're commutative and they're inputs. It's sort of doesn't matter what order the data happens to arrive. You compute the same result. That was convenient in the old days for databases because it meant you could do these very efficient batch computations. Not caring about the order in which the data was represented, but it occurred to us that in distributed systems, it's very hard to control the order in which things happen. It's very hard to make sure that a bunch of replicas get messages delivered in the same order, for example.

If you could remove that the program running on those replicas was insensitive to the order of delivery, those replicas wouldn’t need to coordinate. That is to say, if the replicas were executing a query with certain constraints, then that replication logic is sort of eventually consistent without running Paxos in front of them or something crazy like that, right? The idea was what if we gave programmers a language to write distributed systems. Sure, the secret was they’re query language, but we didn't tell people that.

What if we gave people a language in which the natural mode was to say nothing about order? Just talk about how you're supposed to take input and combine it to produce output? Every now and then allow the program if they needed to, to say, “This has to happen and then this happens.” That was the idea of saying it's disorderly programming. Because a language like C, a language like Java, a language like Erlang, almost any language you can think of, order is kind of the first thing you do. Sequencing is the most fundamental construct in like an imperative language. Do A, and then when you're done doing A, do B. If there's a data dependency between A and B, you kind of need to do A and then B. But if there isn't, couldn't you kind of have done A and B in parallel? The answer is sometimes. Sometimes if you did them in parallel, it would have been fine. Sometimes if you did them in parallel, you’d have a race condition. In an imperative language, the way you fix that is you do some mutual exclusion. You figure out what can run at the same time. You figure out what can’t. But that's art. That's really hard.

As I've said in other talks, when you're doing critical sections, it's this game where if you make the critical sections too big, your code is probably right, but it's slow because you have no parallelism. If you make your critical sections too small, your code is probably fast, but it probably has races. This is a real pain in the ass to program this way. What if instead you gave programmers a language where it was hard, almost impossible to specify order? Then most of the programs they wrote would just be commutative. That's the idea behind disorderly programming. Then if a program decides they need order, because like, for example, they’re trying to implement a lock, and you can’t implement a lock without saying you'll grant the lock until after the other guy releases it. Well, we give them some special constructs to talk about order, but then the idea is when a programmer in a distributed system talks about order, we should make them pay attention, because that's the hard thing. That's the thing that if you get wrong, maybe you have a race, you have a bug. That's the thing that, to make it happen in that order, you might have to pay severe costs. You might have to have a lock server. You might have to run around of consensus.

[00:08:34] JM: I get what you're saying it, the critiques of programming in the traditional imperative ways around data management and programming languages. Those seem like insurmountable problems though. I mean, how are we going to program without explicitly talking about what's concurrent and what has to be serialized? How could we create an environment where it's possible to avoid doing that?

[00:08:59] KS: I mean, I think you might be surprised. Very often, irrespective of the programming paradigm that we're in, we carry out design patterns that are supposed to minimize sharing and minimize conflicts.

An example that was really inspiring to us in the early days of this disorderly programming research project was the Dynamo keyvalue store, Amazon's keyvalue store. Are you familiar with Dynamo?

[00:09:22] PA: Sure.

[00:09:23] JM: Yeah. We have this nice property that although there's massive concurrency in the site, there're lots of users all interact with the site at the same time. We have this really nice property that simplifies it, which is that every user interacts with their own cart. There're no conflicts across multiple users on the cart. To come up with a conflict, you'd have to imagine a user interacting with the cart on their laptop and also on their mobile phone or something like that. We've already kind of designed our system that we're trying to minimize conflicts. Then we go another layer on it and say, "Well, the thing about a shopping cart is you could represent it as just this mutable thing that gets side effected on and every single order in which a change to the cart happen might matter.

For example, if I'm adding and deleting items from my cart and some replica perceives those operations, individual additions and deletions of items to the cart in the wrong order, we might disagree at the end of the day about the contents of the cart. Because if you saw a deletion for an item before you saw the addition, you might think the item belongs in your cart. Whereas if I saw the addition and then the deletion, I think the item is not in my cart, right?

Of course, when you think about it, irrespective of the language you wrote it in, if you just kind of popup a level of abstraction and model the cart as a set, I have a set of things I've added and I have a set of things I’ve deleted, and sets have the property that their API is completely commutative. No matter what order I see the things get added to the card at the end of the day after I’ve union them all up into the set of added items, that will be all the added items, and similarly with the deleted items. When somebody goes to say check out if I’ve modeled my added items as a set and my deleted items as a set, all I have to do is sort of do a set minus of the two to figure out what's like actually in my card at checkout time.

By playing that trick, sure, it's all written in C++, but by playing that trick, you’re kind of modeling this higher-level data type that has this commutative interface. For example, if I gave you the Bloom language, which only, at least to a first approximation, still you start reaching for the ordering constructs, only talks about sets. It will be like three lines of Bloom to implement the Dynamo shopping cart and then you would get the guarantee, “Hey, this code is commutative.”

That’s not to say like that’s a trivial example, but there are many examples like this where the smart engineering implementation, despite the fact that it was written in C++ so you don't get any guarantees, involved abstracting away from the order in which memory things are written. Trying to reason about some application-specific property like I get all the things in my cart. This is not to say you never need a lock, but smart programmers are always trying to design systems where you don't use locks except when you need them.

The idea of disorderly programming was what if I gave you a language where the natural mode, the easy mode was to write the stuff and never needed a lock and you could just write and write and write, and then eventually maybe my static analysis comes back and says, “This code is incorrect. This code is –” I haven’t talked about this yet, but this code is non-monotonic. It's not a query in some sense. Then the static analyzer should tell you, “Hey, you need to go back and add some ordering here.” I never finished the story, the whole idea of disorderly programming.

Traditional programming, orders everywhere. Every single time you put a semicolon, you're saying this has to happen before this. The problem is orders everywhere, but only some of the order is necessary. The necessary order is a needle in a haystack. In a disorderly language, you almost never use order. When you talk about order, you have to do so explicitly. In a disorderly language, you write and write and write and everything is commutative. If something becomes not commutative, the compiler tells you. It tells, “You this line, you might need to add a lock, for example.” That was the idea. You program without order, and the infrastructure pushes back, as supposed to you program with order everywhere.

[00:12:56] PA: Got it. I think it's worth discussing at this point. Your level of interest in applied distributed systems. So there're people who stay mostly in distributed systems theory. Proofs, for example, theoretical things, even TLA+ kind of falls into this category. It's a little more applied. But then there are people who are more practically motivated. They like to play with the cloud tools, the distribute databases, the serverless functions. Do you have a particular interest in one or the other of these?

[00:13:30] PA: First let me say that I look up to the distributed systems theory community immensely. That source of people who publish paper and conferences like PODC, the Principles of Distributed Computing. I'm very influenced by their work and they’re some of my greatest heroes. It's the kind of thing where like I’m still a young professor. I feel like I'm not worthy to be included among. I'm a little bit more applied. I sometimes wish I were a better theoretician. I was an engineer. I'm just over the hill from Silicon Valley. I have a lot of contacts in the Valley. I try to make a lot of my inspiration for the projects we take in my lab come from practical pain points that people are feeling at the companies where I consult and with whom I collaborate, because in large part, I do kind of believe, and this is not unique to distributed systems. This is true in academia, generally, that we’re all smart people. We’re great at inventing problems. If left to our own devices, we’ll probably just invent problems and come up with elegant solutions to them and then it’ll never have any impact, because those problems don't reflect the real worlds. I think it's really important it periodically re-ground in real practice.

That said, so it's very important to stay connected with the real pain points. On the other hand, I'm not the type of researcher who's into doing incremental work. I do want to ask and answer big questions. I used to joke with my grad students, we don't need to worry about getting scooped, because the stuff we are working on is so wild. Nobody would ever do it. Which seems like it's totally impractical, and I don't think it is. I think it's inspired by real problems, but it's blue sky stuff. Because if I wanted to work on containers and sidecars, I would get a job that paid three times as much right over the hill, right? I want to have fun. I know that you mentioned the protocol repair work that I presented Insider last year, and this is like a great example of something that's very much a practical problem, "Hey, dude. Your chaos engineering thing found a bug in my system, and thanks, but all it told me was that an assertion failed or that a user had a bad time. It didn't tell me how to fix it." You've only given me the tip of the iceberg with chaos testing. Help me fix the bug.

In some sense, that work is very practical, but the answer I came up with is very theoretical. It's like, "Well, if you were to write your programs in this very esoteric language of my own design and they kind of look like first-order logic, then I could tell you exactly what the problem was and maybe even repair your protocol," but then we have to go from that and generalize it and transplant it to something that a company like Netflix or eBay or Uber might actually use, by no means easy, but we have at least one example, lineage-driven fault injection that did that. That the initial prototype that I presented in SIGMOD, you had to write your distributed system in this toy language that I'm the only person who knows.

It seemed like a big pill to swallow, and yet at Netflix we were able to generalize those lessons. The research is theoretical and cosmetic in nature very often, because it's a matter of personal preference, but I try to keep it grounded in the real questions.

[SPONSOR MESSAGE]

[00:16:31] JM: Scaling a SQL cluster has historically been a difficult task. CockroachDB makes scaling your relational database much easier. CockroachDB is a distributed SQL database that makes it simple to build resilient, scalable applications quickly. CockroachDB is Postgres compatible, giving the same familiar SQL interface that database developers have used for years.

But unlike older databases, scaling with CockroachDB is handled within the database itself so you don't need to manage shards from your client application. Because the data is distributed, you won't lose data if a machine or data center goes down. CockroachDB is resilient and adaptable to any environment. You can host it on-prem, you can run it in a hybrid cloud and you can even deploy it across multiple clouds.

Some of the world's largest banks and massive online retailers and popular gaming platforms and developers from companies of all sizes trust CockroachDB with their most critical data. Sign up for a free 30-day trial and get a free T-shirt at cockroachlabs.com/sedaily.

Thanks to Cockroach Labs for being a sponsor, and nice work with CockroachDB.

[INTERVIEW CONTINUED]

[00:17:53] JM: Let's go into this discussion of protocol repair with lineage graphs, this paper that you wrote recently. This was around the question of failure injection, chaos engineering, this deliberate tinkering with your system on a regular basis to ensure that it can handle unexpected failures. When you were talking to large companies in your creation of this paper, and I think your goal was to develop a more structured way of thinking about how to do fault injection and then how to respond to it. What did the large companies that you spoke to have to say about the debugging of distributed systems?

[00:18:37] PA: It's a little too soon to talk about the potential impact of that debugging work on the big companies. Let me give you a little bit of background that may help clarify where this work came out of. The reason that I even had relationships with this handful of companies was the previous project that I worked on. It was very related, right? Lineage from fault injection, and here the idea was all the cool kids are doing chaos engineering. Chaos engineering from 10,000 feet is you're running your system either in production or in staging and you're running integration tests. While you're running it, you're perturbing the system by injecting faults, crashing containers, for example, cutting network links.

The ideas that your system was designed to be fault-tolerant. You assume it tolerates some class defaults, but how do you really know unless you try? Did you configure your replication correctly, and so on? The problem, of course, with chaos engineering is that the combinatorial space of possible experiments that you could make is massive if you have 100 computers or sort of 2 to the 100 experiments you could do by turning off computers. You can't kind of do all of them. So you don't do all of them. You do it randomly, stochastically.

The idea of lineage-driven from fault injection was, "Hey, the cool kids also have really sophisticated observability infrastructure deployed. There're all these traces. What is a trace? A trace is a graphical explanation of a distributed execution. How had distributed execution evolved over time and space? It kind of says at different levels of granularity depending on the tracing, how that distributed execution did? What it did, and sometimes how it succeeded? They're in some sense of lineage? A Dapper trace, a Zipkin trace, a Yeager trace is coarse-grained lineage. It's like how did this execution actually happened?

The idea behind LDFI was, if you kind of observed a system over a while doing what it does both in steady state and also under fault, so some of the traces are like, "This is what the system did when you craft computer one and it nevertheless served video to this user." You look at a lot of these traces and you could build a model of what success looks like. With the assistance of that model, you can hack away at that giant combinatorial space. Hey, I don't need to do experiment A, because my model already tells me what my system would do if I did experiment A. So I'm not going to do it, or I'm not going to experiment C, because I think based on my model of the system, if I did experiment D, I would know how experiment C would fall out. I'll just not do experiment C. That's kind of the idea of LDFI. You prune away at this combinatorial space because you have a model of the system that's based on its externally observable behavior via tracing.

At the end of the day, LDFI is supposed to, in a small number of experiments drive the system into some corner case where some unexpected thing has happened and you've revealed a bug. But if you reveal the bug, as I said a moment ago, what happened? Well, a user had a bad time. A user couldn't stream video or something. User didn't get a ride, or maybe your assertion test said your assertion failed. But that's not a heck of a lot of help, because at the end of the day, you need a root cause, or if you're hostile to the idea of root cause on ideological grounds, which I understand. In some sense, there's no root cause of anything. But there is a bug fix that needs to happen, right? If there's something that's making users not be able to stream videos, some engineer needs to go change the line of code somewhere.

Now, we can agree or disagree about whether that line of code was a root cause, but something needs to find a very precise place and make a change, right? That's debugging. This debugging project really came out of LDFI where the advice I got from companies was, “Okay, great. This integration test failed. Now what? I thought you were analyzing the traces? Can’t you tell me anything more about the root cause?” It was kind of like, “Yeah, you're right. We should be able to.” Using that same input dataset, the traces that we used to build the model that told us what faults to inject, and then we injected some faults and we got lucky. We made the system fall down. Using those same traces, there should be a way for us to explain why the system fell down and ideally explain what needs to be changed about the code to make it not fall down.

It wasn't that companies were like we need this exact tool. It’s that the companies were like, “What now?” It occurred to us that it's all part of the same story. If we can build a model of how a system does what it does, then we can start asking questions like, “Well, here's an execution of the system working on the left, and here's a trace of the system not quite working on the right.” By assumption, they share some structure, right? By assumption, the bad execution was similar to the good execution for a while. They’re plugging along. Then at some point, the bad execution took a wrong turn, right? That’s kind of like the intuition.

If you could tell me exactly where the bad execution took a wrong turn, that might help localize the problem. Think of it this way. You have this trace. It's a directed acyclic graph. The nodes in the trace are events that happened in the computation. They happened at some particular place in the distributed system at some particular time. Edges between the nodes mean this event was caused by this event.

Now let’s imagine I have two of these DAGs and they have a common sub-graph describing the initial prefix of the computation where everybody was happy. This isn’t exactly what the tool does, but this is an intuition. If I was to sort of graphically subtract the bad graph from the good graph, what would I get? I would get the things that were missing from the bad graph. If I look at the first thing that's missing from the bad graph, that's going to be a node in the graph. The first node that was in the good graph's execution that was not in the bad graph's execution.

Now, remember that because these traces are generated by a program that has instrumentation points, every node in the graph has to correspond to a logging statement in your program someplace. If you can go back from that node, this node in the trace was the first good thing that the bad execution didn't do. Then you're like, "Okay, hoist me back or ground me back to the code." Well, presumably, the blog is somewhere in the region of that logging statement. The tool is a lot more complicated than that, but that's the initial insight.

We have these graphs, because the graphs were created by the code, if we do graph differencing, we can localize issues and then we can ground them back in the code. Now, in some cases this just works, because in some cases, what you did wrong was you wrote the wrong line of code. You are like off by one, or you wrote something wrong in your exception handler. In that case, my student, Leonard, can actually go back to the code and put a window on the code and say ,"Somewhere in this window, you need to fix something. Somewhere in this window, there is a bug." Because it was this window of code that produced that node in the graph that we've reasoned backwards from. That's the easy case where you've made what you might call a sin of commission. You wrote the wrong thing.

The hard case, of course, is the sin of omission. This code is buggy, because you didn't write all the lines you needed to write. We spent a lot of time on that in the paper and in my talk. You have a protocol and it doesn't work, not because you're off by one or because you fat fingered a config or something like that. There's no error to point to, because the code is wrong because you didn't sufficiently developed a protocol to work around some contingency. The only fix is to like add lines of code.

Amazingly, amazingly, the student's prototype will, in great many cases on real bugs, identify the missing lines of code and supply them. We have this demo, we have this primary backup replication thing. Are you familiar at all with primary backup replication?

[00:25:42] JM: Do you just mean like a database backup from a separate database?

[00:25:44] PA: I just mean like the high-level primary backup protocol. I'll just sketch it really quick. It's a pretty common thing. It could be a database that used it. It could be almost anything. You could just be modeling like a single write or a register using primary backup. But you say it's a primary, and all reads and writes go to the primary and there are some number of backups. The discipline is when a read comes in to the primary, the primary can just go ahead and serve it. If a write comes into the primary, the primary sends it's to all the backups and doesn't acknowledge the client until it has acknowledgments from all the backups. That way, we know that we could failover to a backup and there won't be any lost data.

Now, very often, ingenious junior engineers look at protocols like this and say, "Hey, that's really high latency waiting for all the backups. Why don't we just say, hello and acknowledge the client right away while concurrently sending the matches to the backups. That'll usually work, but there are executions obviously in which if the messages to the backups are lost due to a network partition and then the primary goes down, then we have the situation where the client got an acknowledgment, but the data is gone. That's bad.

This is a pretty common bad protocol to see in practice. What you might call an asynchronous primary backup where the primary acknowledges too soon. This is one of the use cases we put through this tool that my student built called Nemo. In this case, again, the error isn't a misconfigured line, a typo. It's not an off by one. It's the absence of logic that waits for acknowledgment from backups.

How could you possibly automatically fix that? You have to like read the programmer's mind. In fact, what the tool does is adds a few lines of protocol that makes the client wait for acknowledgments from all the backups before considering the protocol to be done. Now, how the hell did he do that? The trick in this case was we were required both the program and the correctness specification to be given to us in a formal language that we could analyze. You didn't just submit your primary backup program. You also submitted the correctness specification. The correctness specification said something like what I just said, primary backup is correct if whenever a client receives an acknowledgment, that data is on at least one correct node. That's a specification node.

Now, obviously, given the protocol that those asynchronous primary backup and acknowledges the client too soon, and given that specification, there's at least one execution. I already kind of described it where the backups are partitioned away in the primary fails where the specification is not the upheld. We have a trace of an execution where we didn't uphold the spec.

How do you repair a program? Well, one way to repair it would be to change the spec, but that's cheating, right? You can’t change the spec. The only thing you can do is kind of change the program until it more closely resembles the spec. Now, the fact that we force the programmer to give us both the program and the spec in a logic language, in a provenance enhanced language, it means that we can record the trace the lineage, the provenance, not just of the program.

Provenance of the program might answer questions like why is this replica in this state at this time as a consequence of the messages that they have received? Those are the kinds of questions that program provenance answers. But we can also ask for the provenance of the correctness specification. When I run the program and everything works, I can take the correctness specification and say, “Why was this true?” and then I get an explanation there, “Oh! this was true because there existed at least one replica in which the write was on.”

I have this graph describing why the correctness specification was true in some execution and have this graph describing why it wasn't true in some other execution, and now I can begin to think about adding edges to a graph, right? To make the bad graph look like the good graph in graph land – And I know this is getting really abstract. Maybe all I have to do is add a few edges among the nodes. If an edge between nodes in the graph when I ground that back in the program, that's just a couple of additional rules in the program.

The way that my student, Leonard, did this was he made sure that he could ask not just why the program did what it did, but why a particular correctness property was true or not. When it wasn't, he could ask what minor changes to the program that really in the abstract graph base were just the addition of edges could force the program to always make the specification true?

In this particular case, going back to the intuition, the fix was making the acknowledgment happen a little later. Adding a couple of additional rules to make the acknowledgment wait until we had heard from everyone, and then he was able to prove that when that is true, there is no way for the correctness specification to be violated.

There are two things, right? The first thing is just differencing, which is a neat insight in itself, that if you can graphically represent executions, you can compare the graphs and use that to localize and find a mistake that you made. But the really cosmic thing is, because you have a graph of the specification, you can begin to ask these questions like what could I add to my program to make the program look more like the spec? Because at some level, a program in a stack both describe the same thing, an input-output relation. It’s just the program has a lot more detail than the spec.

[00:31:01] JM: It makes sense. I know that a specification for a distributed system is really going to be a guiding light in the implementation of that distributed system. It's actually hard to reason about the correctness of your system in practice if you don't have a specification to reach for. Having a tool to help you develop one, even if it's developing a specification in reverse, like you've already got your system and then you’re developing your specification for the system after building the system. That does seem useful.

I understand if the material to come back from the companies the you've worked with on this or the companies that you’ve talked to about this is a little early, but do you see how this could be potentially usable in some kind of workflow? What would be the workflow for actually deriving the specification for your distributed system based on a set of distributed traces?

[00:31:55] PA: I think this is a really great question. I want to make a small correction to something you said, but I think the spirit of what you said is really great, and I hope I have a good answer for it. In this particular case, I was assuming, and it was a strong assumption, and that's kind of how I want to tackle this question, that the specification was given and that it was a precise specification. I was not assuming that we were going to figure out the specification over time. That’s sort of a separate problem, but it's the thing I want to talk about now, because I think this is one of the hardest pills for my collaborators to swallow.

Okay. There's this slide that I give at the end of this talk where say, “Okay, now I've convinced you that I can rewrite your program and read your mind.” That was really crazy. Let me come clean about all the crazy stuff that I had to assume to make it work. I had to assume that you wrote your program in my crazy language. I had to assume that I could stimulate that distributed execution on my laptop. Then the third one was I assume that someone had written down a precise logical specification of what it means for the system to be correct and I didn't just fall off the turnip truck. I know y'all don't have those, unless you're already doing TLA, in which case you're doing TLA. I think that's just a really important question that I have to ask.

From my perspective, to transplant this work successfully to industry in the way that in my opinion I did transplant lineage-driven fault injection successfully to industry, the two main challenges, you just articulated one, are going to be getting precise logical specifications of what it means to be correct. I do think, as you kind of already indicated, that there are some hope here, because we have our assertions, if nothing else. We have our integration tests. That's a starting point, but that's just sort of a binary thing, like the thing worked or didn't. What is the path from the arguably somewhat primitive signals that we have that the system is working correctly and saying that looks a little bit more like a TLA specification in the form of an implication. If a client got a thing, then it is always the case that there exists this, that that. I'm very interested and I'm open to ideas about how we go from existing software quality practices and continuous integration stuff to possibly inferring something that looks a little bit more like a first-order logic specification. I think that a specification inference is like a really interesting thing and a very challenging thing in practice.

I think the other huge challenge is the granularity of the tracing infrastructure. In my walled garden where you're writing these programs and data lists, every single line of the program is a node in the graph. Every deduction that the deduction engine makes is an event in your distributed graph. It's extremely fine-grained, and that fine-grainedness helped me a lot. In the differencing case, maybe it doesn't matter because I do graph differencing. I put a window on your code. Maybe if the tracing is coarser-grained, the window is bigger. But I'm trying to actually do program repair, I kind of need to know exactly what lines to add and exactly where to add the. If you're tracing your system with a shallow coarse-grained call graph tracer like Zipkin or something like that, then hoods in the graph are like, "This service received this request." That node in the graph might correspond thousands of lines of Go code. Not one line. That's like the other challenge, is granularity.

I localize a bug and then I bring you to the code. In my demo, the code was like one line. But at eBay where you're doing Zipkin tracing, the code might be a whole service, right? If I tell you, "I've localized your code." The problem is this service, you're like, "Yeah, thanks. I knew that. It was a alerting which line of code." From my perspective, that's the other challenge. What is the path to finer-grained tracing? If we can't go there, what other signals can we incorporate to do better? Isolation only helps if you isolate to something that a human being could cognitively sort of hold in their head. It's a great proof of concept, but I do assume that the tooling is there, the signal, the tracing has to be there. I can only find my keys under the spotlight so to speak.

[00:35:51] JM: As far as your research compared to the applied failure testing in the form of chaos engineering, like a company like Gremlin has. Do you have any thoughts on how failure testing should fit in to the development of a system whether that's some academic project or a company? How should failure testing fit into your system and how should you respond to the failures that you discover with it?

[00:36:22] PA: I don't know. The landscape is changing all the time. I'm a huge fan of Gremlin. The CEO Gremlin and I worked together at Netflix on LDFI. I love that work and I love the emergence of this community. I think one way in which my work is just totally different in spirit from all the chaos engineering work that originated at Netflix and now is kind of in a diaspora everywhere, is that a lot of the chaos engineering stuff, even from the very inception, it was about a culture. It was about creating a culture of resilience among programmers. Knowing that your services were randomly going to crash meant that you wrote your services in a different way. I think that's so cool.

Obviously, now cultures change over time in morph and split, but I think the idea of creating a culture of resilience where, "Look, distributed systems are uncertain." So kind of turning up the uncertainty dial maybe makes people dot their I's and cross T's a little more, and that really has nothing to do with what I'm trying to do, right? The idea of saying there should always be this background radiation of stochastic perturbation. That's great, but it's not like something you study in the lab, right?

For me, I was always in this much more walled garden where I'm like, "Forget about testing and production. I can't get my head around that." What if I have this test and I can run the test over and over again and I want to be able to reproducibly figure out a bunch of faults that make the tests fail? In that sense, I'm in a much different frame of mind than a lot of the chaos stuff.

On the other hand, as chaos engineering has emerged, and I've been looking for partners and people to fund me and things like this. One of the ways in which I market lineage-driven fault injection in the context of chaos is that chaos is given has given the tools. Right now, you can just assume that large-scale systems are being traced and have some kind of fault injection capability. Given those two assumptions, that's great. You can look at your system and you can perturb your system. But as I said before, the combinatorial space is giant, which faults do you inject? The way I would pick LDFI is that's where I come in. LDFI is an experiment selector.

You’ve got this chaos infrastructure. How do you drive it? You could drive it randomly as Netflix did with the monkey. You could hire chaos engineers who are supposed to be experts of this kind of thing and they just ingeniously navigate the space, but they're expensive and those processes are not repeatable. So the way I would sell LDFI is wouldn’t it be cool if you had some kind of artificial intelligence that was like a chaos engineer that watched your system evolve over time and came up with hypotheses about something that might break it that it's never seen any evidence that it can tolerate. In my model, I've never seen you tolerate X, Y, and Z. So that's probably the next thing I should do. I’ll do it. See what happens.

There're two things, right? If injecting faults in X, Y, and Z crashes the system. Woohoo! We found a bug. Let's go fix it. If it doesn't, guess what? We get another trace, and that trace tells us how the system works even in the face of X, Y, and Z falling, which means we have a richer model of the fault-tolerance over a system, which means next time we’ll suggest something even crazier. Do you see what I’m saying?

That was the idea of how LDFI and my stuff might fit into chaos, is in an advisor role. Consuming the traces, producing chaos experiment selections alongside the random ones and the human-driven ones that people are already kind of doing in the state-of-the-art. That's where I like to imagine that my stuff could fit in. It makes assumptions about the tracing and the fall injection infrastructure just being there and it kind of connects the two.

[SPONSOR MESSAGE]

[00:39:58] JM:

[INTERVIEW CONTINUED]

[00:39:57] JM: Vettery makes it easier to find a job. If you are listening to this podcast, you are probably serious about software. You are continually learning and updating your skills, which means you are staying competitive in the job market. Vettery is for people like you. Vettery is an online hiring marketplace that connects highly qualified workers with top companies. Workers and companies on the platform are vetted, and this vetting process keeps the whole market high-quality.

Access is exclusive and you can apply to find a job through Vettery by going to vettery.com/sedaily. That's V-E-T-T-E-R-Y.com/sedaily. Once you are accepted to Vettery, you have access to a modern hiring process. You can set preferences for location, experience level, salary requirements and other parameters so that you only get job opportunities that appeal to you. If you have the right skills, you have access to a better hiring process. You have access to Vettery. So check out vetter.com/sedaily and get $300 signup bonus if you accept a job through Vettery. Vettery is changing the way that people hire and the way that people get hired. Check out vetter.com/sedaily an get a $300 signup bonus if you accept a job through Vettery.

Thanks to Vettery for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:41:34] JM: Another domain that I would consider related to the practical failure testing that we're discussing is the Jepsen test, which is an effort to improve the safety of distributed systems. What does the Jepsen test encompass?

[00:41:50] PA: I'm really glad you brought this up. First of all, Kyle is a friend of mine. He's also a collaborator of mine. He and I actually have the paper in flight right now that's applying Jepsen related stuff to testing database systems. It's very much in the mix, and I would characterize what Jepsen is, in large part, what Kyle does, is yet another example of state-of-the-art expert-based testing. He's not exactly doing chaos engineering. But if you kind of squint your eyes, he's sort of is. He's not looking at giant scale systems. He's looking at data management systems that are distributed, but we’re talking distributed on the order of 3 to 7 nodes as supposed to thousand nodes, like they're doing in chaos engineering.

But like the chaos engineers, he has a fault injection suite that he uses to inject faults and then run assertions and try to force the database into a violation of its correctness properties. In some sense, he is, if you kind of squint your eyes, the scale is a little different. He is looking at distributed systems but at a much smaller scale.

The definition of what it means to be correct is a little different, because at Netflix, all I care about is availability, for example. Is the site up or down? Whereas Kyle cares about correctness. He wants to show you that your database, it was supposed to be serializable, but was it really? But if you ignore those distinctions, it’s really the same thing. He's a genius. He studies the system. He runs the system. Figures out how it does what it does. He comes up with hypotheses about something that it might not tolerate. He tests the hypothesis by actually injecting faults, crashing nodes, injecting partitions, injecting delay, screwing with clocks, whatever it might be. Then he observes the effect of that and sees if it violates an expectation.

In that sense, it's really the same think. When I developed LDFI in 2014, the original prototype was called Molly. The paper actually says that I was inspired by Kyle Kingsbury. The paper actually literally says something like LDFI is an attempt to capture the unique genius of Kyle in software. He was very much an inspiration for all this work, because I was like, “Well, how does he do what he does?” He watches the system. He builds a model of how it tolerates faults, and that model tells him about the faults that it might not tolerate.

From my perspective, it's great work. But like chaos engineering, and maybe I’m flattering myself or boasting, but I imagine a future where he could do even better by allowing some of my software to make decisions about what faults to inject rather than doing it himself, because there's only one of him. How do we reproduce Kyle Kingsbury? I don't know.

[00:44:13] JM: You’re saying he has these tests that are a little bit manually generated and have him on the critical path and you would like to see a world in which testing some new distributed database was more automatable.

[00:44:26] PA: I'm not saying it's possible, but I'm saying it's a good dream, right? Yeah, yet exactly what you said.

[00:44:30] JM: Right. Cool. What have you learned about the effects of programming language choice on the bugs that ended up emerging in the distributed system?

[00:44:39] PA: This is such a hard question, because you have to understand that, on the one hand, as a professor, I'm constantly telling people that language doesn't mean anything. Once you've learned three, it doesn't matter. It's about how you model your problems. I'm a professor, so I always say that. Languages don't matter.

On the other hand, I spent a big chunk of my life selling languages and telling people that you're programming in the wrong ones. I think it's certainly true that choosing the right language can often help protect us from the worst in ourselves. You choose a functional language in its going to force you to rethink how you share state and how you do iterative computation. In exchange for that mental burden, it's going to force you to do some patterns that are better, right? Better in the sense that it reduces the cognitive burden. But at the end of the day, you don't want o program everything in a functional language. You're going to have the right tool for every job, but I think it is true that when we have a job and then we pick a language, one of the things we should be doing is what are the hard things about this job? The right language for a particular domain is the language that allows us to ignore the things that don't matter or aren't hard in that domain and that forces us to focus on the things that do matter and are hard.

Going back to the disorderly programming thing, order is the thing in distributed systems that's hard. It's where the cognitive burden of the programmer should go up to 11. So we created a language where when you talk about order, you got to think really hard. But most of the time, you don't have to think about it. It's also the case, and this is something that I think about a lot. Take my example about query languages. What's so good about query language?

One of the things that are good about them is their lack of expressivity. They're not Turing complete. They're far from Turing complete, and the fact that there is less that you can say means there's more that you can prove. I talk about this in a recent KubCon talk, there's this very small language. It's a sub-language of query languages called the language of conjunctive queries. The language that you get if you just use select, project and join, for example, in a database. You can't write every program you never want to write in this very constraint language, but you can write some programs. You'd write like a webserver. But the nice property that you get if you write stuff in the language of the conjunctive queries is that program equivalence is decidable.

If I have two programs, one that's three lines and one that's a million lines. I can put them into a decision procedure if they're written in the conjunctive query language and it would say these programs are equivalent or not. If they were, then I could throw away the million-line program and run the three-line one. That's amazing. But if I make my language even the tiniest bit more expressive than that, equivalence becomes undecidable.

This is very theoretical, but it's an illustration of how often the choice of a language is about asking for something that constrains what I'm allowed to say. I can't say something bad or so maybe after the fact some kind of analyzer can actually prove that my program has some property. I wouldn't be able to analyze a C program and be like, "Dude, this program is commutative no matter."

But a Bloom program, I can analyze and say this program is commutative, in large part because you can't say as much in Bloom as you can say in C. I would say you choose a language on a task-by-task basis and it's the language that is just big enough to do the task and not any bigger.

[00:47:41] JM: What are you researching right now?

[00:47:43] PA: I'm working on something with Kyle Kingsbury. It's going to take a lot of time to do the long version, but the short version is I would like to, from outside, a database. I'm just a client. All I can do is connect to a database, issue some writes, issues some reads. See what comes back. I would like to just on the basis of those externally visible observations form sort of a theory of mind about what's going on inside the database so that I can prove if the database has violated a particular correctness guarantee like serializability or snapshot isolation or read committed.

This is hard in general, because from outside the database, you don't get the complete picture. If I read variable X and I get back the number three and you had written the number three to variable X. Does that mean my transaction happened after yours? Well, only if you're the only person who wrote the number three, right? There are all these tricks that you have to do to be able to, given these fairly shallow external observations, reason in a very precise way about what must be going on inside the database. It's that kind of reasoning that's sort of required to be able to prove that the database has failed to uphold one guaranty or other. This is one project that I've been working on.

Then another project that I'm really excited about is the collaboration with one of my grad students, Daniel Bittman and his co-advisors, Ethan Miller, and Darrell Long, who are operating system and storage system researchers. This is like a little bit outside my area, because it's low-level OS stuff. We've been trying to ask him questions about how do we have to rethink operating systems as byte addressable nonvolatile memory begins to become available and cost-efficient?

You can already buy from Intel DIMMs that fit in your DIMM slot that you can do loads and stores on that have the property that they're almost DRAM speed and they're persistent. They survive across reboots. This means we kind of have to rethink everything. We shouldn't be accessing data with the involvement of the kernel at all, because doing a kernel crossing is 50 times slower than doing a read of our memory.

A long time ago, we could hide the expense of the kernel crossings because the disk was thousands of times slower, and we could hide the overheads of serialization and deserialization as you're moving things from memory format the block format for the same reason, because you have so much time to do compute while you're going out to the disk. But if we're going to be accessing persistent data at memory speed, we really need the kernel to completely get out of the way.

What's more, data in this persistent memory is always going to outlive the actors that operate on it, whether those actors are like threads and processes or hold computers. The data is going to have a lifetime that in the limit is eternal. In the limit, the lifetime of processes and threads is going to be nothing. So we're kind of rethinking the operating system from the bottom-up. In the old days, the first-class citizen of the operating system was the process and data was this ephemeral thing that passed from one process to another in a message if you wanted to share. We're kind of turning that whole thing in its head, unsurprising, because I'm a database person. Data should be the first-class citizen of the operating system, and processes are just these ephemera that come along for a second. They're going to instantiate it on your computer or my computer. I don't really care, and they pick up some data and they transform it into some other format.

We imagine a world where distribute this operating system, you can start to ask questions like, "Okay, I have a data reference and it's not a virtual memory reference, because a virtual memory references is just tied to my process, and my process is going to go away." So we have permanent pointers in the data that point to other data that might or might not be local. I might have some data that has a pointer to some data that's on your computer and I want to compute on that date. What do I do?

Well, in the traditional world I would say, "Oh crap! I need some way to dereference this pointer that's on my machine to go scoop the date off your machine and bring it to my machine. Put it in cache so I can compute on it." But in this world where everything is data and data is long-lived, you don't need to look at it that way. You're like, "Okay. So I have this pointer that you have, but I also have a program, and that's just data." So I have this program that wants to as operate on that pointer. It almost doesn't matter where it happens. The rendezvous of the program and the data could happen in Illinois if that happened to be the place that had the resources.

We can imagine shipping my program to Illinois and your data to Illinois, and then Illinois has a compute resource that's free and it starts – It's like total disaggregation, right? It starts computing. Well, guess what. It's compute state is also data. There's a stack point and an instruction reference, and that's a data object too. So that computation could be suspended and opportunistically shipped somewhere else with those three pieces where the computation could resume. I mean, it's totally crazy vision, but it's really fun to be applying some these ideas I've had much lower in the system stack. We have an ATC paper coming out about this idea and we have a paper in flight to symposium on cloud computing about it as well. I can send you the references if you're interested. As you can tell from how fast I'm talking, it's very new and exciting stuff.

[00:52:18] JM: Yeah, very cool. Yeah, please do send me some links about that. That'd be interesting. Well, Peter, thank you for coming on the show. It's been great talking to you.

[00:52:24] PA: Same. I really appreciate your excellent questions. This was a lot of fun.

[END OF INTERVIEW]

[00:52:38] JM: The Uptake is a new show about all things tech and community. It's hosted by Anna Chu who travels the world of technology uncovering people's journeys, and each episode showcases communities around the world helping listeners understand the value in community activity and exploring the different ways that they can stay up-to-date and explore different ways of doing things with technology. Every episode has a focus topic, guest perspectives from Microsoft MVPs and community news. There're also updates on events and conferences, virtual conferences these days and more. You can find it by just searching for the Uptake in your podcasts.

[END]