EPISODE 483

[INTRODUCTION]

[0:00:00.6] JM: When you are writing code, you are manipulating objects. You might have a user object that's represented on your computer and that user object has several different fields; a name, a gender, an age, and when you want to send that object across the network to a different computer, the object needs to turn into a sequence of ones and zeros that will travel efficiently across the network. This is known as serialization.

As the user object sits on your computer, it is represented in ones and zeros and you could just send that same representation over the wire, but we use efficient serialization to send it over the network in a more compact format. We also have to make sure that when we send that object to another service, the other service knows how to de-serialize it and turn it back into a format that we can operate on at the application level.

Protocol buffers are a serialization protocol that originated at Google. Protocol buffers created a standardized interface for efficiently passing data between services. When Kenton Varda worked at Google, he was the tech lead for protocol buffers and he joins the show to explain how proto buffs work and a newer serialization protocol that Kenton lead called Cap'n Proto. You can expect to walk away from this episode with an understanding of how serialization protocols work and the design trade-offs that you can make when you are creating your own serialization protocol. We also touched on a startup that Kenton founded called Sandstorm and how he eventually found himself at Cloudflare where he now works on the Cloudflare workers.

With these topics, we did not go as deep as I would've liked, but we did cover proto buffs in significant detail. I look forward to having Kenton back on in the future, because there was plenty of stuff that we could have gotten into and we will in the future.

[SPONSOR MESSAGE]

[0:02:05.1] JM: Today's episode is sponsored by Datadog, a cloud scale monitoring and analytics platform. Datadog was built to bring clarity to complex dynamic applications in the cloud, on-premises, in containers or wherever they run. With beautiful dashboards, seamless

integrations with more than 200 technologies and distributed request tracing, Datadog provides deep end-to-end visibility into the health and performance of modern applications. Visualize key metrics set alerts to identify anomalies and collaborate with your team to troubleshoot and fix issues fast.

Try it yourself by starting a free 14-day trial today and listeners of this podcast will also receive a free Datadog t-shirt. You can get all those things by going to softwareengineering daily.com/datadog. That's softwareengineering daily.com/datadog.

[INTERVIEW]

[0:03:05.7] JM: Kenton Varda is the former tech lead for protocol buffers, the lead developer of Cap'n Proto and the founder and lead developer of sandstorm.io. He's also a systems engineer at Cloudflare. Kenton, welcome to Software Engineering Daily.

[0:03:19.6] KV: Hi. How are you?

[0:03:20.6] JM: I'm doing great, and I'm looking forward to talking about data serialization and some different strategies for doing that. Let's start off with the basic question; what is data serialization?

[0:03:32.8] KV: Well, you have a data structure and memory in your program and usually it's doesn't exist in contiguous memory, because you might have like a tree of data or a map of data that you're modifying, removing parts adding parts, but when you want to send that over a network, it needs to be contiguous bytes. You need some way to take the data and pack it into bytes to send it and for the other side to unpack that into data structures useful for computing on.

[0:04:03.8] JM: We can communicate over a network by sending messages in JSON or XML, but if we're just sending those messages, we need to have some way of sending those over the wire. I mean the messages over the wire get put into ones and zeros eventually. How does JSON, for example, get serialized and sent?

[0:04:30.2] KV: You invoke a JSON encoder. You might call JSON.stringify in JavaScript and you give it a JavaScript object, and the coder is going to iterate over all the fields of that object, and for each one produce some text, that's field name: value, and if the value is another object then that's going to be recursive, puts itself in braces. I guess that's basically it.

[0:04:57.5] JM: Yeah. No. That's totally fine. That's a good explanation. I think the lesson here is that this is a seemingly simple problem. You just need to objects into ones and zeros so that we can communicate them over the wire, but there's a lot of ways to turn an object into ones and zeros and we can do that with — The knobs we are going to tune are going to depend on — If we're communicating from JavaScript to a Go service, for example, the way that a JavaScript application consumes an object that is in JSON might not be the same way that an application in Go wants to consume that object in JSON. Why do different programming languages have different ways of representing objects?

[0:05:45.4] KV: Yes. This is interesting. In JavaScript — And JSOn was very much designed around the JavaScript object model. JavaScript is a dynamic language. So you don't have to know at compile time what fields a particular object might have. It's just all figured out at runtime, and that works really well for JSON, because when you parse a JSON object, this message coming in over the wire could have any field names that the sender sends.

In JavaScript you naturally get this JavaScript object and now you can try to access certain fields, and if they're not there, you'll get an undefined value back, and if you try to use that it'll probably throw an exception. Whereas in a type safe language like Go or C++, it actually ends up being a lot less convenient because you need to convince the compiler that you are accessing data that's really there. You end up having to call some sort of function saying like, "Get this field. Here's the name," and then it might return a value if it's there or it might return null and you have to check that.

Interestingly, JSOn ends up really inconvenient to use this often in these type safe languages unless you have some sort of a system for pre-generating code around that, which is a lot of what proto buff does.

[0:07:05.4] JM: Great. That's a great introduction to protocol buffers. They were first developed in Google around 2001. You just alluded to part of the problem that they were solving. What exactly was the intention of the proto buffs or otherwise known as protocol buffers? What was the intention of that project?

[0:07:22.5] KV: Well, there are a few different problems that Jeff and Sanjay, the creators of protocol buffers and so many other technologies at Google were trying to solve initially. One problem is that — So they had fronted servers that were talking to backend index servers in the search engine and every time they made a change to the protocol, added like new kinds of data that needed to be returned, they ended up with this code on the other end, on the receiving end, that would have to do a lot of if statement saying, "If version equals X, or if version is more than X, then expect this. If not, then expect this," and it became a huge mess.

Really, what they wanted to do is have a way to be able to naturally add a new field and have — Well, so I should say part of the problem here is that the they can't — When you have a large distributed system, you cannot just deploy or update all of your system at once. You might make a change to the protocol and then update the backends, but the frontends are still running an older version, and so they have to be able to communicate with the newer backends and be able to ignore the data they're not expecting, the new stuff that was added. You might — You're not going update even all of your backends at once. You're going to do a rolling update if you don't want any downtime, and so they all have to be able to talk to each other and understand each other despite being at different versions.

Protocol buffers, the main thing it's trying to do is deal with that, and it does that in a pretty simple way, which is you define your data structure, which is it's like a struct. You have a bunch of fields. Each one has a type and a name, and you can add a new one, and if a message is encoded that includes this new field and is sent to an old server, the old server ignored it.

Going the other way, if a new server receives data from an old server that is missing a field, that field has a default value that is reasonable and allows the new server to do something reasonable. That was the main problem they're trying to solve. There are a couple of other problems solved at the same time. One is efficiency. I should say that the versioning problem is actually solved pretty well by JSON as well. Like, in fact, it's almost the same model. JSON

doesn't have a way for you to like predefine default values. You have to actually do it in the code. You have to check if the field is there. If not, user default. That's pretty easy to do. Generally, JSON has the same model for version upgrades.

Proto buff also wanted to do a couple of other things. One is — So when you're encoding data into text and then reading that back on the other end and turning it back into data structures, the computer is doing a lot of work just to make it so that the data sent over the wire can be read by humans. But 99.99% of the time, no human is actually going to look at that. We're spending a lot of CPU time. Operations are basically not useful most of the time.

Protocol buffers instead uses a binary format that is much faster for a computer to put together and to parse at the other end. So that's the second thing it does. The third thing I would say that proto buff does that's really useful is the generation of code, code generation. So you define your data structures and then you input them into the protocol compiler and it generates classes for you in your desired language that make it easy to manipulate the data structure you've defined. That allows you to do things like define a default value and have the gather method for this field will automatically return the default if the field is missing instead of you having to write the check yourself.

It's also when you have a binary format, you need a parser. The parser can be really fast if the code is generated instead of operating dynamically based on tables. It's an optimization, but more importantly it gives you this type safety. It makes sure that you can't — If you misspelled field name, you won't get an error at runtime. Your compiler can catch that assuming you're using this type safe language, and to me that's the most important thing, is just the increased ease of use that you get out of that.

[0:11:55.3] JM: You've just defined so many different reasons for the existence of protocol buffers. If we just talk about the simple example of a Node.js service, which is a JavaScript communicating with a Go service. We've got JSON objects that need to be sent from the JavaScript service to the Go service. If we were doing this naïvely, we would just send a JSON blob. We would just naïvely serialize it and the Go service would be responsible for de-serializing it and figuring out if all of the fields are intact figuring out its own getter methods, its own — Yeah, its own getter methods to pull the data from the objects that got sent over the wire.

It would have to figure out, “Okay. What version is this object? Did it come from an old API version? If it came from an older API version, maybe it doesn't have certain fields or it has extra fields that I don't need to worry about.”

Basically, protocol buffers are these — It’s just a way of making communication between different services easier to work with by giving you typing. You're typing these objects, which in in JavaScript they might not have a type, and if in a typed language, a type may might be required. It gives you a getter. It puts them in a format that is going to be more compact than if we were just naïvely sending them over the wire.

There's all these different things that we get out of protocol buffers, and the way that we use protocol buffers in one of our services that we want to communicate with another service that uses protocol buffers is we define a schema, and the schema is the thing that translates our objects into these proto buff, the serialized proto buff objects that will get sent over the wire. Can you explain a little bit more about the schema? I think this is also called interface definition language.

[0:13:55.1] KV: Yeah. With protocol buffers you have these files, these .protoprofiles. Some people call them IDL. Some people call them schemas. Some people just call them protos rather ambiguously, but in it you define your message format. So in proto buff you define a message type. It's a lot like defining a class in an object-oriented language. You define the set of fields that it has. Each one has a name. Each one has a type and each one has a number, and the number is important for compatibility.

You can change the name and not affect compatibility of the messages sent between old and new servers, but if you change the number, then will break. Usually, there's not really much motivation to change number, whereas people like to rename things all the time. This is actually one of the reasons this works so well and people don't accidentally introduce breakages, it’s because of these numbers.

There’s such a thing as JSON schema, which is a similar thing for JSON. People rarely use it. I'm not entirely sure why. Once you've defined this schema file, this proto file, then that's what

they input to the protocol compiler that I mentioned earlier, and then it outputs code in your favorite programming language based on that input.

[0:15:13.0] JM: There are many instances where data gets sent between two places. So if I have a micro services architecture, different services are requesting data from each other. If I have a more simple web app, the user might just be making a single request to a server and loading in their browser. If we're talking with these two different communication patterns where services are talking to one another versus a client just talking to a server, would I use proto buffs in both of these types of communications or is this just for services communicating with each other?

[0:15:48.7] KV: This is a matter of debate. One thing about trying to use proto buff in the browser is that it can be inconvenient. It can add a lot of code to the JavaScript that has to be downloaded up from, and so a lot of people just choose basically to use JSON there, because JSOn is pretty convenient to use from JavaScript. There is already a JSON parser built in to the web platform, and what you can do is on the server-side that your browser talks to, you can have a converter layer that basically converts JSON into proto buffs or other formats. Most of them have some sort of conversion library that you can use.

Also, like another point is that on the browser side you have lots of CPU time to work with. Now, when you get into sort of the server in big distributed systems where you're crunching a lot of data, that's where it starts to matter if you're spending 30% of your CPU time just encoding things and you want to cut that down.

[SPONSOR MESSAGE]

[0:17:03.4] JM: DigitalOcean Spaces gives you simple object storage with a beautiful user interface. You need an easy way to host objects like images and videos. Your users need to upload objects like PDFs and music files. DigitalOcean built spaces, because every application uses objects storage. Spaces simplifies object storage with automatic scalability, reliability and low cost. But the user interface takes it over the top.

I've built a lot of web applications and I always use some kind of object storage. The other object storage dashboards that I've used are confusing, they're painful, and they feel like they were built 10 years ago. DigitalOcean Spaces is modern object storage with a modern UI that you will love to use. It's like the UI for Dropbox, but with the pricing of a raw object storage. I almost want to use it like a consumer product.

To try DigitalOcean Spaces, go to do.co/sedaily and get two months of spaces plus a $10 credit to use on any other DigitalOcean products. You get this credit, even if you have been with DigitalOcean for a while. You could spend it on spaces or you could spend it on anything else in DigitalOcean. It's a nice added bonus just for trying out spaces.

The pricing is simple; $5 per month, which includes 250 gigabytes of storage and 1 terabyte of outbound bandwidth. There are no costs per request and additional storage is priced at the lowest rate available. Just a cent per gigabyte transferred and 2 cents per gigabyte stored. There won't be any surprises on your bill.

DigitalOcean simplifies the Cloud. They look for every opportunity to remove friction from a developer's experience. I'm already using DigitalOcean Spaces to host music and video files for a product that I'm building, and I love it. I think you will too. Check it out at do.co/sedaily and get that free $10 credit in addition to two months of spaces for free. That's do.co/sedaily.

[INTERVIEW CONTINUED]

[0:19:22.2] JM: Now, if I am starting up brand-new project and my requests are fairly small and they're fairly infrequent, do you think it is a premature optimization to use proto buffs there? You're hacker. Do you start with proto buffs in place for all of your basic applications that you build or is there a volume of data throughput where I should start to use proto buffs?

[0:19:47.5] KV: I have not used proto buffs in a long time, because I use Cap'n Proto these days.

[0:19:50.9] JM: Okay. Fair enough. A serialization framework or whatever you would want to call these.

[0:19:54.3] KV: Yes. I actually very much do like to start a new project by writing out schema files before I write code, because it's basically the interface. I should note on the last question, like even though the browser side client may be sending JSON up to the server, I'm still going to write a schema file and it may still be in proto buff or Cap'n Proto format and then I use the converter library that I mentioned.

But I like to start with these schema files because it lets me think about the interface and the interactions between systems before I think about the implementation details. Having it written out, like when you write out the schema file, you're basically — It's just your API. It's not the implementation, and it's a great way to sort of outline the design.

[0:20:49.9] JM: Yeah. I mean it's the API of an object. It's not exactly the API of the service, but I guess it's the interface of the object. Would you say that's accurate? I guess you could define the services around what those objects, what their attributes are.

[0:21:11.0] KV: It's the object that — It's the data format that you're using to communicate to the server or between components. So it really is the API for those components, is these message formats.

[0:21:23.5] JM: Okay. Fair enough. So I want to get to Cap'n Proto eventually, and I think the way to get there is I'd like to talk a little bit about your work at Google. You were the tech lead for protocol buffers, which is a pretty incredible role, because I've been in a lot of different places that use protocol buffers, and you lead the development getting proto buffs from V1 to V2 and getting them open sourced. What was that project like?

[0:21:52.3] KV: Yeah. Well it was fairly ad hoc actually. Early on at my time at Google, I took an interest in protocol buffers. I started adding some features to it, because basically the state of the project was when people needed something, they add something. Then the company was getting bigger and that was getting kind of difficult to maintain just having everyone add what they need, and people started looking to me to maintain the project. Then I said, "Hey, we should open source this." Basically everyone said, "Yeah, that sounds great. Do it."

So I ended up being the one doing most of the work for that, but the state of the code at that time, the proto one code base is very much tied to Google's internals, lots of other internal Google libraries and it was going to be really hard to pull that all apart in order to open source this component. So what I ended up doing was rewriting it. Also, proto one had evolved slowly over time and not everything was planned out in advance, and things get a little messy. So I rewrote it, clean some things up and with the intent of being able to open source it, and then did, and then it took off.

[0:23:04.8] JM: It's interesting to hear the similarity between that and what I hear when I talk to the Kubernetes developers, because they said that when they were building Kubernetes, they couldn't just open source Borg. Borg is the project that Kubernetes was based on, but they couldn't open source it because the code base was tightly coupled with Google infrastructure. So a lot of it just wouldn't even make any sense if they open sourced it. It sounds like that was the same case with proto buffs.

[0:23:30.9] KV: Yes. Google basically has their own set of infrastructure that they've built from scratch. Everything is written inside Google. There're very few external things that they use, and they have enough people, enough really smart people that they've built some really great stuff and it all turns into this nice little internal world that they used to build products, but then when you want to release any of it, it's very difficult.

[0:23:57.6] JM: You saw protocol buffers adopted by large companies like Twitter, and then of course there widely used within Google. Do you remember any anecdotes from especially the external use cases where you talk to people and they explain to you how proto buffs improved to the infrastructure that they were working with. Were there any very specific use cases that come to mind, like, "Oh my gosh! This changed my life having proto buffs."

[0:24:26.2] KV: I feel like I've heard that kind of thing so often that it is hard to come up with specific use cases. You mentioned Twitter. My understanding is that it was a big part of their rewrite for scalability. Although my understanding is they don't use exclusively proto buffs. They use some other serialization formats as well, and I don't really know the details. Yeah, I don't know. I don't know a specific example off the top of my head.

[0:24:55.6] JM: I'm sure it contributed vanquishing the fail well. You left Google in 2013 to start Cap'n Proto which is an open-source project to be the successor to proto buffs. So was this you left to work full-time on the open-source project?

[0:25:12.3] KV: I left to start a startup, which is sandstorm.io. I knew that that's what I was going to do eventually at the time that I left, but I wanted to spend some time playing with some ideas that I have had around proto buffs.

One thing I could change about protocol buffers at Google was the basic underlying encoding, because Google has petabytes of data in this format. It can't possibly change because then they'd have problems reading all that old data.

Now when I left I thought, "Well, now if I wanted to write a new proto buff, it could have a completely different format and I could play with ideas to make it faster." A few people had asked me about this idea of, "I want to share messages in shared memory. If I have two programs that can both access a segment of memory, how do I efficiently allow them to share data through it?" A problem you run into their — Well, if you're using proto buffs, basically what you have to do is you encode your data structure into this memory and then the other side decodes it. It just seems like a big waste to like turn in-memory data structures into an encoded representation and then take it back out all on the same machine instead of just like letting them reference each other's memory.

Another problem that — Proto buff is much more efficient than JSON, but there are still a lot of services inside of Google and other places that use a lot of CPU time just encoding and decoding proto buffs. I thought that if we could change the underlying format so that it was closer to what the in-memory data structures would eventually look like, we could save a lot of that time. Especially when you're talking about services inside of the data center, talking to each other, network bandwidth at this point is basically infinite between two machines on the same record in the same data center. Proto buffs spends a lot of time encoding integers in a way that makes them variable length so that smaller numbers take fewer bytes, and it's all just to save few bytes on the wire. That's kind of wasted when bandwidth is not your problem and CPU time is.

[0:27:32.4] JM: You're describing the same issue that I talked to the Apache Arrow people about. I don't if you've heard of that project, but it's sort of — For example, if you're doing "big data" and you want to do some crazy, like hot high-volume data processing between Python and Hadoop, for example. Like you're doing some data processing pipeline where you have a Hadoop job and it's a big map reduce and then you want to hand it off to something that's in Python. Before Apache Arrow, you would have to convert that data from a format that Java understands to a format that Python understands, because Hadoop is in Java and Python is Python. You have the same data transference problem that we talked about with protocol buffers.

Apache Arrow is a way of standardizing the in-memory representation. So instead of doing this wasted effort of serialization, you just represent it in a way that both the map reduce job and the Python job can access, and you're talking that whereas Apache Arrow might do that with some large-scale data processing, you're describing something that at a fundamental level it's essentially the same thing except it's like microservices communicating with each other on a frequent basis over the wire so the volume is overtime rather than like in large batches.

[0:28:56.4] KV: Yeah, sounds similar. Actually, I'm not very familiar with Arrow, but it sounds like the same problem. Yeah.

[0:29:02.6] JM: This in-memory representation thing, if we want to have the same in-memory representation in contrast to serializing and de-serializing into our own, I guess, native in-memory representation, what kinds of changes does that require? That's a rather dramatic change between — Contrasting protocol buffers with Cap'n Proto. Cap'n Proto being the shared memory, shared in-memory representation format versus the proto buff's serialization-deserialization aspect. What architectural — And the developer APIs. What changes when you shift that model?

[0:29:43.4] KV: Yeah. First I want to add one more thing about where this is useful. Shared memory is where I started thinking about it, but where it has often come up in practice is you have a large file on disk. Let's say it's many gigabytes of data and it's all encoded as one gigantic message. Now, people would do this with proto-buffs sometimes. The problem is in

order to read any data from that file, you have to read in the entire file, parse the entire thing into an in-memory proto buff data structure and then you can play with it.

With Cap'n Proto, you can use the trick called memory mapping where you tell the operating system like, “Place this file into this address in-memory and don't actually read in the data from the file until I access the memory.” This is a feature that most operating system support and that I think is underused because it's really cool.

If you do that with Cap'n Proto, since the encoded format is appropriate as an in-memory format, which means that it can be randomly accessed, you can then go and read the one part of the file that you wanted just naturally and only the pieces of the file that are needed to support that will come into memory.

To answer your question about the programming interface, so mostly using Cap'n Proto actually looks a lot like using proto buffs. There are some — When you get into the details — When you use proto buffs, as I said, you generate these classes which are in-memory data structures and then you read into them and sometimes people then go another step and convert those into some other representation that they really want to use now.

Now, Cap'n Proto also generates classes for you. The trick is that all of the accessories in these classes, instead of reading member variables of the class, they do pointer arithmetic on an underlying buffer and figure out where to read that data out of, but that's all hidden from you, because you’re calling these successors and it looks a lot like proto buff. Sometimes the API in order to really make the zero copy aspect work, there are some spots that end up a little bit more awkward. For the most part, it’s the same model.

[0:32:07.2] JM: Does that mean that you have to write stuff that’s specific to each programming language? Because we talked earlier, like if you want to convert — If you want to have proto buffs convert JavaScript objects, JSON objects into something that Go can understand, you have to write the serialization protocol for doing that conversion. Do you have to do something like that when you're reading from these — You're doing this arithmetic to a pointer arithmetic to understand how to read different objects in different — Sorry. The same object in different

languages, because you're talking about in-memory representation that can be accessed by, for example, a Java service as well as by a Python service. How do you do that?

[0:32:53.1] KV: So this is all handled for you by the library. There is a — Just like with proto buff, proto buff has ready-made implementations for many programming languages. Cap'n Proto also has ready-made implementations for a fewer number of languages, because it's not as mature, but they're there.

The main implementation that I've written is in C++, but there's a Go implementation. There's a Python implementation and so on. You use those libraries, and they handle all the — These pointer arithmetic happens under the hood. They're dealing with that and you just get a nice interface where you call get field, set field.

[0:33:29.0] JM: Okay. You left Google to start Sandstorm and you wound up working on Cap'n Proto. I want to talk a little bit about Sandstorm and how that informed some of your decisions in building Cap'n Proto Proto. Talk a little bit about what Sandstorm is and what the motivations for building it were.

[0:33:49.1] KV: Sandstorm, it's infrastructure for web apps with a unique design. Product-wise, one way to think of it is it's something like Google Docs or Dropbox where you have a bunch of data that you can collaborate on, like document editors and such, but it runs on a machine that you control. Like you can download this and run it on your own server, and it's all open source, but it's also a platform for these kinds of apps.

So there's an app store, you install apps. So one kind of app might be a document editor. Another one might be an RSS reader. Another one might be a chat service. The idea was to make it really easy for anyone to do that and to actually run their own server and make it even like just as easy as using these online services was the goal.

At a lower level, the infrastructure is designed in a very different way from the way we view most service today and that every instance of a — Like if you have a document editor app, like Etherpad is a popular one on Sandstorm. Each document you create runs a separate instance of the Etherpad server in a separate container isolated from all the others. This is a thing we call

fine-grained containerization. No one else that I know of really does this, and it has a lot of really interesting properties that come out of it, like that the system, Sandstorm, can now manage access control for you.

So if I share a document with you and then I have another document that's super secret and I don't want anyone to have access to, because Sandstorm manages the sharing, no bug in Eitherpad or whatever the app is can allow you to — Based on your access to the document I shared with you to like hack your way into the other document, because the platform itself enforces that separation. So security. Big focus on security.

[SPONSOR MESSAGE]

[0:36:00.5] JM: Consensys is the largest blockchain company focused on building software on the Ethereum platform. They've developed Truffle, the most popular Ethereum development framework. Truffle is your Ethereum Swiss Army knife and it's available for free by going to softwareengineeringdaily.com/consensys.

Nearly 200,000 developers are working with Truffle and you can download it today and start building your own software on Ethereum. Find blogs and tutorials there as well to get started. Truffle is written in JavaScript in a completely modular fashion allowing you to pick and choose the functionality you'd like to use. For example, you could use Truffle as a library in our own tool using only the modules that you need. This lets you take advantage of powerful features, like Truffle migrations in your own command line tools.

Consensys has built several of the leading dapps, decentralized applications, in the Ethereum ecosystem and offers some of the most popular free Ethereum developer tools such as Metamask, Infura and Truffle. These tools are essential if you're thinking about building an Ethereum dapp.

Learn about Truffle and download it directly from softwareengineeringdaily.com/consensys to get going on Ethereum development. If you want to hear a show about one of the topics that Consensys knows a lot about, send me a tweet @software_daily and tag @consensys, that's

consensys with a Y instead of a U, Consensys, with the topic that you would like to hear about. Tag both of us and let us know the topics that you're interested in hearing about. Thank you.

[INTERVIEW CONTINUED]

[0:37:51.4] JM: To put a finer point on what you said, Sandstorm is a way of — It's an app platform where you've got things like Etherpad, which is this collaborative Google Docs type of document store. You've got a Trello-like task and project management system. You've got a Dropbox-like file storage system. You've got a chat system that is somewhat like Slack, but the difference is you manage the data yourself and you have control over it, and it's open source so you actually know what the code is. Am I articulating it correctly?

[0:38:28.8] KV: Yes. I think currently all of the apps are open-source as well, and these are — By the way, we didn't develop these apps. These are all separate apps. Some of them were developed specifically for Sandstorm. Some of them were existing open-source apps that we made work on Sandstorm. Yeah, gathering all these together and making it easy to use them.

[0:38:51.0] JM: Okay. Well, now I'm regretting not doing more research about Sandstorm before this episode, but I guess I had enough content just from the proto buffs and Cap'n Proto stuff. Yeah, okay. I'll opportunistically jump into that. So you built this infrastructure that allows people to build apps that are self-managed, but I guess they can also sync with other instances. I guess is there like a peer-to-peer way of resolving conflicts? For example, like we have a shared document on Etherpad and I have the copy and you have the copy and we want to synchronize that. Conflicts can occur, right? Did you build all the infrastructure for resolving those types of conflicts?

[0:39:37.1] KV: The way that works today is one person is the owner of that document, and the document lives on their server. That basically largely solves that kind of problem in a straightforward way.

[0:39:51.0] JM: Actually, that's really interesting, because — It's so funny, because I talked earlier today and I think I was telling his before the show about CRDTs to Martin Kleppmann, and he was talking about CRDTs as a way of resolving conflicts in a system like Google Docs,

for example. The way that Google Docs does it is they funnel all the conflicts through a central server, which is not exactly a decentralized model.

You're talking about a decentralized model where like you and I are controlling our data, we're controlling our own documents, but you can still do the — I think it's called — What is that? Operational transform way of resolving conflicts, but it's fine because the two of us controls the server where those changes are centralized.

[0:40:39.4] KV: Right. Etherpad, for instance, is an example of an app that uses operational transforms. In fact, they invented operational transforms.

[0:40:48.4] JM: Wow! Okay, interesting. Talk more about what the infrastructure of Sandstorm is. What are the APIs and stuff that you've built?

[0:40:57.7] KV: Yeah. It's interesting, Sandstorm is one of the few pieces of infrastructure where the end-user is actually expected to interact with the infrastructure as part of the whole experience. You log into your Sandstorm instance and it gives you a list of your files or your documents or your chat rooms or whatever, which we call grains. They're fine-grained instances of apps. Then you choose one to open and then it opens in an I-frame within the sandstorm UI, and Sandstorm provides some of the features like sharing and file management while the app itself provides the actual business logic of this particular app.

The big challenge in making that work is — So now there is this — Apps aren't any longer responsible for a whole lot of these sort of boilerplate things, like how do I do access control and how do I do file management and such. Instead, the app is now focused just how to render its particular type of data. While that makes it easier to develop apps in this model, a lot of existing apps already had built a lot of these things and now they kind of have to delete them, delete all that code in order to fit into Sandstorm and integrate with the Sandstorm APIs Instead.

The Sandstorm APIs — So the sandstorm — Bringing you back to Cap'n Proto here, the underlying communication layer between all the app instances is Cap'n Proto based. In fact, when an HTTP request comes in and is destined for a particular app, actually it hits the Sandstorm frontend proxy first, which converts it into an HTTP over Cap'n Proto format in order

to send it to — Route it to the appropriate backend, and there's a lot of reasons we did that, but I could go on for too long.

[0:42:50.1] JM: In terms of Sandstorm, do I host a version — So if I'm hosting my own Etherpad, for example, where I want to do document management and I've shared that document with you and I am the — We decided that I am the source of truth. You Kenton and I are sharing this document and this document is within Sandstorm. Do I have a server instance? I have a sandstorm server instance running on my local box, and then HTTP — If you make a request that you want to change that document, you send in a HTTP request to my Sandstorm server. Is that correct?

[0:43:31.3] KV: Yes, and more broadly like I would visit your servers, like the Sandstorm UI on your server and then I would open the particular document that you've shared with me, or you might send me a sharing link which would go to the Sandstorm UI on your server with that document opened within it. Yes, HTTP requests for that I-frame, which displays the app. The HTTP request for that also go to your server, but go through this proxy and end up at that particular app.

[0:44:00.6] JM: Got it. Now, when you make that request and it hits my Sandstorm server, you said it's transformed into a Cap'n Proto representation of the HTTP request, and then what about — On my Sandstorm server I'm going to be running an app like Etherpad or I'm going to be running an app like the Trello version on Sandstorm, some project management system. So is there a shared memory representation where if I want to handoff that HTTP Cap'n Proto representation from the Sandstorm HTTP conversion into Cap'n Proto, if I want to hand it off to my Etherpad instance, do I need to transform it at all or does the Etherpad instance just read directly from that Cap'n Proto format?

[0:44:55.1] KV: It depends on the app. Most apps on Sandstorm use a tool we call Sandstorm HTTP bridge. It's a little program that actually ends up being the main process of your app, and then it runs — You give an HTTP server application that it runs as a child process and then the bridge receives the Cap'n Proto requests and converts them back into HTTP over loopback, but there are some apps that directly implement the Cap'n Proto interfaces, and those ones tend to be very fast and quick to load.

[0:45:34.7] JM: We've talked about a lot of different concepts here. Can you talk more generally, like if somebody want to — Let's say somebody has their infrastructure built around protocol buffers and they are considering switching to Cap'n Proto for their service-to-service communication serialization protocol. How would they evaluate those that type of migration? Would you want to make that kind of migration or is this for different types of use cases?

[0:46:07.9] JM: Yes. It depends on how much stuff you have. How many protocols you've defined? How many types you have? If you have hundreds of proto files all defining lots of different types. Doing a full migration to a different format is probably impractical. As software engineers, we have to live with that sometimes. Sometimes we're stuck with something that isn't as good as it could be, but we move on, because it's not the worst problem that we have.

Now, if you have a format or protocol that's defined in like one schema file and you have a few types, then you can pretty easily evaluate. You can try writing the same schema in Cap'n Proto and then writing the code both ways and then you can actually do a benchmark and see which one is faster. I would only go to this effort if you see CPU profiles showing that you're spending a lot of time serializing and de-serializing. If so, then it would be worth evaluating between the two.

Just to note. Don't trust benchmarks done by someone else on data that isn't your data, because each of these formats has strengths and weaknesses and it completely depends on what kind of data you are sending which one is going to be fastest. I don't actually publish — I have benchmarks for Cap'n Proto versus proto buff and I'm happy about where they are, but I don't publish the numbers, because it's not really meaningful for someone else to make a decision based on those. You really have to test your own setup and see how it performs there.

[0:48:02.0] JM: Okay. I know we're kind of running up against time, but I want to talk a little bit about Cloudflare, which is where you now work. What was the transition from working on Sandstorm full-time to Cloudflare? What motivated you to go back to kind of big company world?

[0:48:20.9] KV: Well, we ran out of money. Sandstorm was a startup and it received funding, but ultimately the business model that we really need to go for was one involving a lot of enterprise sales, and that's hard. In classic fashion we underestimated how hard it would be.

Last December or so we looked around to — The company was going to have to shut down and we looked around to see where we wanted to go, and I was really interested in Cloudflare, because I've always — Sandstorm was a Cloudflare customer. Well, I guess technically still is a Cloudflare customer, and I was always impressed by their focus on security, their focus on infrastructure. I'm an infrastructure engineer, I like being somewhere where infrastructure is the product where it's not like when I was doing infrastructure at Google, it was a lot wait for a product team to ask for something and then do what they ask. Whereas at Cloudflare, it's actually the thing that we're selling and I like that a lot.

[0:49:30.8] JM: Now I can — I'm sure that was a really painful experience and I think I can relate to it somewhat, maybe not to the same degree, but for the last year I I was working on a company called Adforprize. I spent a lot of money and I spent a lot of time on it and I didn't end up shutting it down. The product still exists. I would love to revisit in the future. At some point I'm sure you think about the same thing with Sandstorm. You didn't like shutter it down, because there's no reason to. It's like probably is not super expensive to run, or I could be wrong, but it's probably not massively expensive to run, or at least at a minimum host the open source code out there. There's no reason to delete that from the world, but it's funny because I mean what you said, you underestimate how long it's going to take or you underestimate what the sales process is going to be and eventually you have to just have this reckoning moment where you just have to admit to yourself, "Man! It sucked, but this not going to work out the way it worked out in my dreams."

[0:50:38.1] KV: Yup. That's pretty much what happens. I still work on Sandstorm an an open source project. I'm still making updates and I intend to continue doing so. There's still a community publishing apps, building things. We recently added, for instance, internationalization and now there's a bunch of people translating the Sandstorm interface into a bunch of different languages, which is really cool.

Yeah, I don't know where it will end up if it will just continue to be an open source project. Just is the wrong word. That's a great outcome, or if it might become a business again someday. I don't know.

[0:51:16.9] JM: Yeah. Has there been like some solace in going back to a place where now you wake up in the morning and you know that you work and you're going to be remunerated for that work on a reliable basis. You know that your work at Cloudflare — For example, I know you work on Cloudflare workers, is going to have massive impact. Have you found some solace in the fact that you're going back to working on stuff that a large customer base are paying for and really treasuring?

[0:51:52.3] KV: Yes, exactly. So Cloudflare was definitely the right place for me to go. I've had a lot of fun there. This project Cloudflare worker has actually started from scratch when I joined Cloudflare. I'm the lead of that, and it's been a lot of fun being able to write a new service from scratch, making all the design decisions the way I want to and not have to worry about things like; how am I going to sell this? Because there is already a sales division of Cloudflare that can help me with that.

Knowing that this is going to be — It's pretty clearly going to be a big deal once we get this out the door, but then just being able to focus on the engineering part, which is the part that I do well and the part that I enjoy is really great. Yeah, getting a regular paycheck is pretty great too.

[0:52:41.6] JM: Yeah. I know it's impossible to explore Cloudflare workers in the level of detail that I would've liked to, because I probably managed our time not as well as I could've, but just to explore it a little bit. At a basic level, Cloudflare is a giant cash with 100+ locations around the world, and cloud flare workers is a way to push custom logic to edge servers. Basically, it's a way of making your edge computing and your cash servers operate more intelligently because you can deploy programmable logic to those edge servers. Some of us — Maybe you could call it serverless computing, because you sort of deploy the logic and you're not exactly managing a server. You're managing the notion of your caching and your edge computing. Maybe talk a little bit about what the goals of a Cloudflare worker are and how it contrasts with other serverless types of systems?

[0:53:47.0] KV: Yeah. I think the future of cloud computing is that you don't have a central server for your applications anymore. Instead, you send the code to wherever it's best for it to be running. If you have code that is operating in some particular database, you probably want to send the code to run next that database, and I think we'll see a lot of big database services start letting you run code directly on them in the future, bits of JavaScript for instance.

Now, Cloudflare, the thing that Cloudflare does is it's close to the users. 90% of the world's population is within 10 milliseconds of latency to a Cloudflare location. If you want to have code that can respond very quickly to end-users, you put it in a Cloudflare worker. Actually, this could end up being the place where most of your logic ends up going in the long term, because a Cloudflare worker is arbitrary JavaScript. It's using the service worker's API, which is an existing W3C standard API that exists in browsers today, but this runs on Cloudflare poplar servers and it basically gets HTTP request in and then it responds to them however it wants, and in the process it can make sub-requests to other servers not just to your own server, but any server. So you can make your API requests from the edge and assemble your response. Maybe do your HTML templating on the edge and return that and eEventually end up with a better experience for users, because they're not round-tripping all the way back to your central location for every request.

[0:55:26.0] JM: Well, that sounds like it ties in nicely with your notions of efficiency that led you to starting Cap'n Proto, "Let's avoid these round-trips. Let's avoid these excess serializations. You're doing a great job to make the Internet infrastructure more efficient. That seems like a great place to wind down the conversation.

Lastly, do you have an example, like any case studies for people who are really leveraging Cloudflare workers or how somebody in the audience might be able to leverage them?

[0:56:01.2] KV: Yes. Here's a really common thing. You have a website, say, you're news service. You have a website. It has a bunch of content on it that's all very cacheable. As long as people are not logged in and they're visiting your site, they get everything served out of cache that's 10 milliseconds away from them. But then as soon as they log-in, because you want to offer subscriptions or whatever, now you want to write at the top of every page like, "Hello such

and such. You are logged in. Manage your subscription options," but now the page is different for every user. Now, you can't utilize the cache anymore.

With Cloudflare workers, what you could do is add that name to the top of your site on the edge. So you're actually loading the public cache content and then you're modifying a little bit using the — The user's name might just be in their cookie, and so there's no need to make a request back to your application server at that point. You can do it all on the edge and then you're utilizing the cache much better, you use much less bandwidth and response times are way faster.

[0:57:09.5] JM: Okay. That's a great place to wind down the conversation. Kenton, I want to thank you for coming on Software Engineering Daily, and all your projects are really interesting. It was great talking to you.

[0:57:19.1] KV: Thanks. You too.

[END OF INTERVIEW]

[0:57:22.5] JM: At Software Engineering Daily, we need to keep our metrics reliable. If a botnet started listening to all of our episodes and we had nothing to stop it, our statistics would be corrupted. We would have no way to know whether a listen came from a bot or from a real user. That's why we use Encapsula to stop attackers and improve performance.

When a listener makes a request to play an episode of Software Engineering Daily, Encapsula checks that request before it reaches our servers and filters bot traffic preventing it from ever reaching us. Botnets and DDoS are not just a threat to podcasts. They can impact your application too. Encapsula can protect your API servers and your microservices from responding to unwanted requests.

To try Encapsula for yourself, go to encapsula.com/2017podcasts and get a free enterprise trial of Encapsula. Encapsula's API gives you control over the security and performance of your application and that's true whether you have a complex microservices architecture, or a WordPress site, like Software Engineering Daily.

Encapsula has a global network of over 30 data centers that optimize routing and cacher content. The same network of data centers that is filtering your content for attackers and they're operating as a CDN and they're speeding up your application. They're doing all of these for you and you can try it today for free by going to encapsula.com/2017podcasts, and you can get that free enterprise trial of Encapsula. That's encapsula.com/2017podcasts. Check it out. Thanks again, Encapsula.

[END]