EPISODE 1064

[INTRODUCTION]

[00:00:00] JM: Data steam is a word that can be used in multiple ways. A stream can refer to data in motion or data at rest. When a stream of data is in motion, an endpoint is receiving new pieces of data on a continual basis. Each new data point is sent over the wire and captured on the other end. Another way a stream can be represented is as a sequence of events that have already been written to a storage medium. This is a stream at rest.

Pravega is a system for storing large streams of data. Pravega can be used as an alternative to systems like Apache Kafka or Apache Pulsar. Flavio Junqueira is an engineer at Dell EMC who works on Pravega. He joins the show to talk about the history of streamed processing and his work on Pravega.

If you enjoy this show and find it useful, you can help us out by help us out by subscribing. You can become a paid subscriber at softwaredaily.com/subscribe and it's $10 a month or $100 a year and you get access to all of our old episodes without ads. That's over 1,300 episodes and there's lots of content in there on anything that you're learning right now.

Also, at softwaredialy.com, you can find question and answer and all kinds of other content relating to this episode which can help you augment the knowledge that you're going to learn from today's show.

[SPONSOR MESSAGE]

[00:01:30] JM: When I'm building a new product, G2i is the company that I call on to help me find a developer who can build the first version of my product. G2i is a hiring platform run by engineers that matches you with React, React Native, GraphQL and mobile engineers who you can trust. Whether you are a new company building your first product, like me, or an established company that wants additional engineering help, G2i has the talent that you need to accomplish your goals.

Go to softwareengineeringdaily.com/g2i to learn more about what G2i has to offer. We've also done several shows with the people who run G2i, Gabe Greenberg, and the rest of his team. These are engineers who know about the React ecosystem, about the mobile ecosystem, about GraphQL, React Native. They know their stuff and they run a great organization.

In my personal experience, G2i has linked me up with experienced engineers that can fit my budget, and the G2i staff are friendly and easy to work with. They know how product development works. They can help you find the perfect engineer for your stack, and you can go to softwareengineeringdaily.com/g2i to learn more about G2i.

Thank you to G2i for being a great supporter of Software Engineering Daily both as listeners and also as people who have contributed code that have helped me out in my projects. So if you want to get some additional help for your engineering projects, go to softwareengineeringdaily.com/g2i.

[INTERVIEW]

[00:03:19] JM: Flavio, welcome back to the show.

[00:03:22] FJ: Hi, Jeff. Thank you.

[00:03:24] JM: When we started this podcast about five years ago, there was a common pattern for building applications around large streams of data, and in order to get us to a conversation about the present, I want to start with this time in the past and that was this time of the Lambda Architecture. The Lambda Architecture was this pattern where you would have fast, real-time data streaming into your system and that fast real-time data would be handled by streamed processing. Streamed processing was great. It was getting popular around that time, but you would occasionally lose packets of data. This process of sometimes lossy stream processing could be reconciled periodically with a big batch process. Why was the Lambda Architecture popular for data applications?

[00:04:19] FJ: One reason it was popular, it was because there tools that enabled people to implement that kind of architecture. But one thing that I find interesting about the whole

discussion of the Lambda Architecture is that the main point that at least I extracted from the original post that Nathan Marz wrote was not so much about the architecture itself, the split between speed versus slow. It was more about the immutability of data.

The fact that if you have a source where you can – Not a source actually. If you have a repository of data and you can keep your whole history of data there, instead of having mutable data, records where you are just updating place. If you keep a record of the new data you're generating and you're able to reprocess all that, then it's easier to reason about systems and even gives you the ability of talking about the history of the records and how they evolved into a given state.

That post from me was more about that or that whole idea of Lambda was more about that. I think the fact that we went into this split, these two paths where you have one compute that say with Hadoop, right? Hadoop MapReduce coming from say HDFS and another one using – I don't know, say Storm, was mainly because Hadoop was not really designed to give you low-latency. If you want to really process the data in this way where you have your whole history of data, say, read only or append only history where you're reprocessing every time you need to. You wouldn't be able to do that with low-latency.

Having a second path, a speed path was essentially a patch to the technology that existed at the time. That's in fact the main observation that came sometime after when Jay Kreps talked about the Kappa Architecture. Immutability was very, very important, but there was no reason why you really needed to do these two paths, right? This slow path and this fast path. If you have a source that even stores data in S3 manner, you should be able to go back in time and reprocess if you need to. There is no reason why you have to limit yourself to just the tail of the stream let's say.

[00:06:38] JM: The Kappa Architecture, this was an architecture that was built around a buffering system and I'd just like a description of the Kappa Architecture from your point of view.

[00:06:54] FJ: From my point of view, the Kappa Architecture or the idea in the description of the Kappa Architecture is that you can have your history of data stored in a log form. Every time you want to make a change to your data, you log a new record to that append only log and if

you want to process a data, if you want to derive results out of that, you process that log. You can do two things. One is you can tail that log, essentially processing the new stuff that is coming. At the same time if you need to process for whatever reason it is, I think if people refer to accuracy a number of times, that's not necessarily the only reason you would want to do that. There are other reasons like – I don't know, you change your code. There is a bug in your code that you fix. You want to reprocess data. You can just go back in time in your stream and reprocess it. Actually, not stream. You refer to it as a login.

Using stream and logging, append only logging interchangeably here. But the idea is that you have this sequence of data changes let's say, which is immutable, and as new data comes in, you're just appending each of the log. If you need to derive any results of that, if you need to propagate a database, an output database let's say that you want to query upon, then you process that data in any form that is suitable to you, right? Tailing the log or reprocessing from some arbitrary point in time.

[00:08:29] JM: I think if we contrast these two strategies, the Kappa Architecture and the Lambda Architecture, the Lambda Architecture was these two distinct systems. You've got the real-time stream processing system where the data is coming in quickly, but it may be unreliable. Then you got a Hadoop-based batch system that is going to have very reliable but extremely slow path to being materialized as a view into your application, and the Kappa Architecture replaces all of these with a change set of all of the events in your data architecture and those events were often being written to Kafka. Then the different consumers of those streams of events could materialize views however they saw fit.

I think one thing that was valuable about the Kappa Architecture is just this decoupling. You just have a central data stream where people can build applications off of it however they want. What I want to mention here is something that confused me for a long time about streamed processing, which is the fact that if you think of a stream, you think of something that is moving. In some cases, that is what's going on. Like you might have a stream of data being ingested and every time you're receiving data at your endpoint from a mobile device that let's say a streaming location data to you, you're getting this data. It's kind of in-motion you might say.

But there's this other notion of a stream which is the abstraction of a stream that has been written and is maybe continually being written to, which is like a data structure or you might say a storage format, which is a stream of data like historically. It's a historical abstraction, a stream of events that has been written. Could you help me unpack this what I think can be a confusing notion the fact that a stream is not always like a moving piece of infrastructure.

[00:10:43] FJ: A stream, we can characterize a stream as parallel sequences of events that are being ingested. They are either coming from a data source. You can think of – I don't know, say many sensors in an IoT application that are producing samples, right? Data, that could be an example. You could also think of some derived data out of processing. That is data you ingest and you're processing it and you're producing different product flows of a data out of your own processing. All those are possible when you talk about streamed processing.

That ingestion of data, when you are receiving that data and you are writing, say, to your log, you can keep that for as long as you want. There is no reason why you should keep your streamed data temporarily. As you're saying, this is in motion. There are a number of use cases for which it's true, like typical pub-sub cases produce publish consume. You publish data. As soon as you consumed it, and consume is an application, is application notion, right? An application decides when it has consumed it and it acknowledges back. So it's done. Then you can get rid of the event or the message or whatever you have in the system.

That would be more along the lines of what you're saying. But there is another class of cases, of uses cases, where you don't necessarily want to get rid of your streamed data or even the sequence with which you have ingested a way. You want to keep that for reprocessing. Again, using the case of say code changes or bug fixes, you process your data, realized that there was a problem and you want to go back in time and execute the same code, the fixed code over the data.

If you don't keep your data in a streamed form, then that will force you to go use a different system maybe, or either you loss a data altogether, in which case you can't reprocess it, or if you just have moved that from a streamed form to some other form, like a set of files, right? You would have to process a data from a different system, which is inconvenient from a development perspective.

Just to complete the thought. In the end, it's a very good point when you say, "I think of streams as dating motion," and that's true. You have an aspect of streams that is about the data that has been generated now, right? Those would be the applications that would say they are tailing the stream and there's also the historical side, which is the same stream data in the same sequence that I still want to store and keep in data in that same form whenever the application wants to process it.

[00:13:34] JM: Okay. The data that we're talking about, the streams that are being created, they're going to need to get written to some kind of system for storage. What are the requirements we have for that storage system for this high-volume streaming data? What do we want out of a storage medium?

[00:13:55] FJ: We want it to be friendly towards append only structures, right? You want it to provide high-performance in the presence of appends. You would like it to be to provide low-latency in the presence of small reads. Events are not always – They aren't always large. Sometimes you can batch, right? You can batch and get higher performance, but you can always do that. In principle providing high performance in the presence of pub-sub events that's also desirable.

[00:14:30] JM: We have requirements for how to read this data and requirements for how to write it. What kinds of failures can occur during the read and write process of a streaming application?

[00:14:43] FJ: Right. Let's go with the flow. You start with put some data source that is generating data, and as you receive that data, you want to write it. After you have written it, then consumer applications can read the data and consuming whatever form makes sense.

Now from the source of the writer, so already there you can have issues. The source may crash, in which case you don't know if you'd be able to recover the events that it has produced or not, right? The source perhaps produce some samples, but it doesn't know it has reached the writer application or not. If the source is able to remember the data that he hasn't written, replay it. Right then you should be able to recover from that fault.

Similar to that, you can have disconnections, right? The connection between the two may drop. Again, you don't know what are the events that have made it or not. You need to take into account that situation too. Now, the next step would be the writing application when it's writing to the storage system. You can also have crashes and disconnections, in which case when it comes back, it needs to be able to determine at which state it was at the time that it crashes so that when it resumes writing, it writes from the last offset or from the last events or whatever abstraction it's using to write. Then the data is stored.

Let's say that we are able to deal with all those faults, like either source or the writing application cashing, the disconnections, and then data is stored. Now on the read side, you also need to be able to deal with the same. The reading application crashing and resuming from the write position or disconnecting and also resuming from the write position. One type of semantics in all these that people typically talk about is exactly one semantics, or there are the three terms that people use, at-least-one, most-once, and exactly-once. Guaranteeing exactly-once is hard, and if you think about exactly-once in trend, you can't really provide it if you don't have application help.

Some of the things that we provide that you can provide so that it can have that exactly-once in trend are first the ability of tracking the position of the writing applications. How much they have been written. In the case, say, they disconnect and come back, they know where to resume from. The other supervised, say, transactions. Transactions where you can determine whether some set of messages or events have been written or not, you can determine that by knowing whether the transaction has been committed.

Another one is to provide checkpoints. If you look at frameworks like Apache Flink, Apache Flink, it has this checkpointing algorithm to be able to provide end-to-end, exactly-once semantics end-to-end. You can do that by providing the ability of checkpointing at the source and transactions at the end of the job graph. If you do that, then with, again, the help of the application, in which case this is the checkpionting mechanism that Flink provides, you're able to provide exactly-once semantics end-to-end.

[SPONSOR MESSAGE]

[00:18:19] JM: JFrog Container Registry is a comprehensive registry that supports Docker containers and Helm chart repositories for your Kubernetes deployments. It supports not only local storage for your artifacts, but also proxying remote registries and repositories and virtual repositories to simplify configuration.

Use any number of Docker registries over the same backend providing quality gates and promotion between those different environments. Use JFrog Container Registry to search the custom metadata of the repositories. You can find out more about JFrog Container Registry by visiting softwareengineeringdaily.com/jfrog. That's softwareengineering.com/jfrog.

[INTERVIEW CONTINUED]

[00:19:12] JM: I'd like to give us a motivating example here for a streamed processing application and maybe go a little bit deeper on the exactly-once issue. In a typical architecture, let's say we have a mobile application that is streaming location data. Maybe it's a ridesharing application. I'm walking around outside with my ridesharing application and this application is periodically sending data to the server about where I am. It's sending a lat-long coordinate system, and that data is probably hitting some backend service like the – I don't know, user location tracking service and maybe that service is writing the location data to the Kafka log or the distributed storage log, whatever kind of storage system I'm using. Can you tell me more about how the data is moving from the user's application eventually into the storage system and what are the issues, the – Maybe you'd call them connection partitions that could occur in the networking layer and what would happen in this streamed processing storage system that could create difficulties around that exactly-once processing?

[00:20:41] FJ: If you have an application, so your user is an application, a mobile one, you also mentioned ridesharing. If you have events that are coming out of that application and you're trying to ingest them into a storage system, you use Kafka as an example. There are other systems that we can think about. There is Pravega, which is the one that I've been working on, then there is Poster, which are other systems that they'll talk about messaging provider, messaging abstraction.

For those systems, things that could go wrong are that one. You lose – The message never gets to this system. In the processing of getting it to the publisher or the writer, to the writing application, that message gets lost because of, again, a network partition or disconnection or a crash where something that happens in that process.

If it never makes to the storage system, then you miss the events. Your log won't have the events. The other possibility is that if you have a disconnection and you don't know if you have sent the event or not, you duplicate the events and you end up having a duplication in your log. These are log typical problems that these log systems or these stream storage systems need to deal with.

[00:22:10] JM: Right. You might have this client that would have to do things like send multiple instances of the event or risk losing the event. In some cases, you're going to have the storage system need to reconcile whatever problems happened at the client layer or in the networking interface between the client layer and that backend storage system.

[00:22:33] FJ: Right. For example, in Pravega, which is a system that I'm familiar with, the clients, the Pravega clients, when it's writing, it informs the server-side the event number that it just wrote. If there's a disconnection, when it reconnects, it gets that event number back and it knows where to resume from. That's one way of avoiding that duplicates are written from the Pravega client perspective, right?

But then if the application, if the application induces a duplicate because the application wrote twice, then internally the client can't prevent that. The application also needs to use mechanisms to prevent that from happening.

One way is to use transactions. You can, for example, create a transaction. You write the events, and if anything goes wrong, you abort the transaction and you start over. In that case, you avoid duplicates. The systems I have mentioned provides transactions. Pravega provides transactions. You have the ability of doing exactly that.

[00:23:45] JM: You've mentioned Pravega a few times and most of the listeners probably know what Kafka is. We've done a show on Pulsar. Explain what Pravega is and how it fits into this spectrum of backend stream storage processing options.

[00:24:03] FJ: We have started Pravega with the idea of building a storage system that is extreme as its main primitive. We're coming from a background of storage systems, so if you think of file systems, object stores, fine objects are well-known storage primitives, but Extreme is an abstraction that we talk about a lot in these times, because a lot of applications are built on data sources that are continuously generating data. It makes sense to think of a storage systems that exposes Extreme as its main primitive.
We departed from that observation, that we wanted to have a storage system that had stream as its main primitive.

Another key observation is that we wanted this system to satisfy the requirements of cloud native applications and a few things that come to mind when I say that are the ability of storing an amount of data per stream. A stream can run for a long time, months, years, and there is no reason why architecturally you're not able to store all that data for a stream and in a streamed format.

The second one is elasticity. If you think about the traffic that is coming into a stream or a set of streams, there is no reason to expect that that traffic is static. It's always the same. That trafficking can vary overtime. You can have periodic cycles – I don't know, daily cycles, weekly cycles, whatever kind of cycles that makes sense. You can have occasional spikes, and you should be able to accommodate those changes.

Elasticity, which in Pravega we provide in the form of scaling, is an important feature. Consistency is already mentioned, exactly-once. The ability of both failing stream and processing historical data, it's not desirable for an application that it has to be with two different systems so that it tails from one and processed historically from another. Ideally, we provide that in a single system. Again, that's well-aligned with typical cloud principles, which are the ones of having an amount of resources that accommodate the requirements of your application.

[00:26:24] JM: Tell me more about the story behind Pravega. I'm particularly interested in how the timeline compares to the timeline of Kafka. When did Pravega get started and how did that relate to what was going on in the broader data engineering community particularly with regard to Kafka?

[00:26:48] FJ: The project itself has started in 2016 and has been going on since 2016. One of our main observations is that it's important for a system that provides the abstractions. I was mentioning about the properties that was mentioning to be more flexible internally and to not rely on storage of the local servers.

In Pravega, we have this notion of stream segments which enables us to do all the things I have mentioned. The fact that a segment that I can break a stream into segments allows me to have parallelism, it allows me to change dynamically the set of segments. It allows me to spread segments across different servers. The fact that I have cloud storage backing Pravega, so that allows me to be elastic with respect to storage capacity. All those things are things that we did not see in other systems, in existing systems, and we have made that our goal to build a system that had all that.

[00:27:52] JM: Go a little bit deeper on how the usage of Pravega might compare to Kafka, because I think – Sorry to position this completely relative to Kafka, but I think people are mostly familiar with the streamed processing and streamed storage usage of Kafka. I'd love to get a little bit more of a comparison in terms of the usage.

[00:28:17] FJ: Right. Pravega allows you to ingest data streams from sources that are continuously generating data and our main goal, as I mentioned before, is to ingest that data and store that data for as long as the user wants it. If the user wants to keep that data, say, for years, stay there for regularly reasons, you need to keep the data for say two years. There is no reason why you should need to move the data from outside your system, say, Pravega, to another system if there is any possibility that you're going to reprocess based on that data.

That's a typical use case of using Pravega as storage. If you look at the hard drive of your laptop, if you store a file there, I mean, assuming that it never breaks, you expect to see that file there forever. The file won't go away. There's not going to be any pressure for your to remove it

unless you choose to. Of course, if you're running out of space, you might decide to clean up, right? That's part of the problem that I'm mentioning. If you don't have a good way of scaling your system so that you don't have to reclaim data as often, then you would end up being forced to delete the data from that system and perhaps moving elsewhere, archive it elsewhere so that you can afterwards.

This idea of ingesting streamed data and keeping it for as long as you like is one of the core idea of Pravega. The other one is the one of scaling. As I was mentioning, Pravega, you have the ability of increasing the degree of parallelism and decreasing it dynamically. We do that today based on the ingesting rate of individual segments. If a segment for example gets hot, then Pravega will – And if the stream configured for autoscaling, then Pravega will transform that into multiple segments. It increases the degree of parallelism for you.

Internally, Pravega has this ability of creating segments easily, sealing segment, keeping the order of segments so that it can have this autoscaling and at the same time be able to have key order. All those are important aspects that we provide and other systems don't.

[00:30:44] JM: Okay. I think I'm understanding. If I use Kafka, Kafka is typically thought of as storage that will eventually be tiered out. You don't keep five years' worth of history in your Kafka cluster. At least I don't think so, because I think a large portion of your Kafka is in-memory. I think part of it gets snapshotted to disk, but it's not thought of as a long term durable storage system. Am I understanding that correctly? The usage of Kafka is usually not super long-term, right?

[00:31:24] FJ: It's exactly, exactly, and this has been a pain point for a number of users and this is something that we have such a solve.

[00:31:33] JM: Right. I think the way that people typically alleviate that is by sending data from Kafka into a data lake? Is that right? Typically, like you'll stream topics to a file in – Like a parquet file, like a parquet file to tier it out of Kafka in order to have some long-term durability to your data. Maybe you could tell me about the workflows that people typically do to make their Kafka data durable.

[00:32:09] FJ: Right. That's one options that people use, the HDFS is a typical option. You take data out of kafka and you write it to, say, HDFS and you keep your data in HDFS files. But one thing I want to mention in that direction is that I think even the Kafka community acknowledges that this is an important feature. In fact, there is a proposal in the Kafka community to provide tiering to offload data to a second tier and that'd be done or handled inside the cluster itself.

There is acknowledgement that building that internally as part of the cluster makes sense, because you provide – In principle, the system will continue to provide all properties that they're used to. One of the key problems is if you use a different system, that system will have a different API, different properties. Guaranteeing that you have a correct system using two different storage systems, that can be complicated.

That's why so we started this in 2016. We had this way of tiering storage and then writing the data there and we're using this trend where other systems are doing this too. Poster added this to their system. Kafka now is talking about adding it. I think there is an acknowledgement from the road community encompassing all communities of these systems acknowledging that this is an important feature.

[00:33:43] JM: If we want to combine the best of both worlds, we want to have a system that can store our more recent pub/sub streaming data and have that data for fast access, but we also want to tier it to a storage layer that is cheaper and more persistent. If we want to combine these two things, is that what you've tried to do with Pravega?

[00:34:16] FJ: Yes. Definitely. That's one clearly one aspect of it. You can tail. We call it is a stream. That's the abstraction we exposed, but if you want to think of that as a log, then you're able to tail your stream or your log, which means that you're reading form memory cache and serving with low-latency. But at the same time, you're keeping that stored long-term. We call it a long-term storage. We keep it in long-term storage so that if you want to process the data historically, you have that ability as well. You're right in your observation.

[00:34:53] JM: Okay. Pravega has these two tiers of storage. There's a tier one layer in Apache Bookkeeper and a tier 2 layer that could either be in HDFS or in an object storage system like S3. Describe this two-tier architecture in more detail.

[00:35:10] FJ: Right. The first tier is used primarily as, say, a journal. When we receive an append an event writer, that would be persisted to BookKeeper. It's written to Bookkeeper. BookKeeper in its turn, it will make sure that it's written to its journal and when you respond back to the Pravega, it's persisted in the Bookkeeper service, which are called bookies.

The segment store in Pravega, which is the name of our server, only then will respond to the event writer. When we respond, when we acknowledge an event, we made it durable. So we guarantee durability for events that are written, and that involves only tier 1 so far.

Now, we asynchronously move the data to tier 2 so that we keep the data in BookKeeper under control, because otherwise if we end up storing a lot of data in the bookies, then we would end up with the same problem as when Kafka was writing only to brokers, right? You have a limited amount of local store and then brokers eventually and the bookies in this case would eventually run out of space. We asynchronously write to tier 2 so that we keep the amount of storage in the bookies under control. Eventually the data lands in the second tier, which is the long-term storage tier and then we could see the data at rest. That's how the two tiers in Pravega interact with each other.

[00:36:47] JM: If I want to integrate this into my data application, does Pravega work similarly work to Kafka? Do I treat it like a pub/sub queuing system or is this more of just a storage format? Could you describe in a little more detail the usage difference between Pravega and Kafka?

[00:37:13] FJ: Right, that's a very good point. Our API borrows concepts from previous systems, like in many systems that I have done in the past, you always try to keep some abstractions that are familiar to developers so that it makes it easier for the developers to understand how to use your new system. There are similar concepts to Kafka as you were saying.

We have the notion of event writers, which is the equivalent of the producers and we have the notion of event readers, which is the equivalent of the consumers and you can form reader groups, and the reader groups, they coordinate internally to do the assignment of segments.

Remember, when I mentioned about autoscaling. The set of segments can be changing dynamically. So the readers in a group need to coordinate. That's one of the features that either group provides. Yeah, that's one aspect in which it's similar, event writers and event readers.

I also want to mention that we have other APIs. We have for example a batch API that allows an application to read from a stream in an ordered manner. If you read directly from the stream using the event stream API, you will follow the order of ingesting. Pravega internally keeps track of the history of segments and it will make sure that it's served following the order of creation of segments.

If you use the batch API, you'll be able to read from all segments in parallel. It won't follow any order, but if for some applications you, that's what you might want to do. For example, if you – It's not necessarily a realistic application, but if you're counting words, you don't care in which order they came. You're just counting the words and it doesn't matter. It doesn't matter necessarily the order. You just want to count them. You have that possibility.

Also, another API that we exposed is what we call the byte API, because internally we do not store events. Event is a concept of the API. Internally, we store bytes. The segments in Pravega are sequences of bytes. To you get from events to those bytes, we need serialization on the way in and deserialization on the way out. But internally, again, we store byte. We have an API which allows you to just write a flow of bytes directly to a Pravega segment. All those APIs are available, but the event-based one is one that is closer to concepts around we made in Kafka that developers should be familiar with.

[SPONSOR MESSAGE]

[00:39:59] JM: Today's show is sponsored by StrongDM. StrongDM is a system for managing and monitoring access to servers, databases and Kubernetes clusters. You already treat infrastructure as code. StrongDM lets you do the same with access. With StrongDM, easily extend your identity provider to manage infrastructure access.

It's one click to onboard and one click to terminate. Instantly pull people in and out of roles. Admins get full auditability into anything that anyone does. When they connect? What queries

they run? What commands are typed? It's full visibility into everything. For SSH, RDP and Kubernetes, that means video replays. For databases, it's a single unified query log across all database management systems. StrongDM is used by admins at Greenhouse, Hurst, Peloton, Betterment and SoFi Control Access.

Start your free 14-day trial of StrongDM by going to softwareengineeringdaily.com/strongdm. That's softwareengineeringdaily.com/strongdm.

Thank you to StrongDM for sponsoring Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:41:21] JM: Describe the typical user of Pravega. What is their situation when they begin using Pravega? Do they already have a Kafka cluster or are they somebody that is just greenfield looking for a streamed processing or a streamed storage system? Help me understand the typical problem that a Pravega user is having.

[00:41:47] FJ: I'm not sure there is a typical user. We have had all of the ones you mentioned. We have the greenfield ones that don't have anything installed and they're looking to moving to streamed processing. They are not necessarily sure what streamed processing means and the tools that are available. There are other ones that already use something else, it could be Kafka, it could be Poster, it could be RabbitMQ, right? Just to mention other systems that are related. For those, they have some background on these systems.

[00:42:18] JM: Let's talk a little bit about the usage more. If I want to connect an application to a Pravega cluster, what's the process for doing that and what happens when I start writing data to it?

[00:42:33] FJ: You can deploy Pravega in a number of ways. The primary way we offer today is with Kubernetes, is deploying Pravega on Kubernetes. We have implemented Kubernetes operators. We have one for the Pravega components, a controller and segment store, and we have operators also for ZooKeeper and BookKeeper.

Following instructions we have in our open source repository, it's easy to deploy it on Kubernetes and get it running. For test and development, there is a standalone way of running it. I mean, of course it's not durable. But for development purposes, it works great. So you can start it locally and then start coding against it. There are other options that we offer too. You can do Docker Swarm. You can deploy the Java processes directly.

Once you've deployed it, then the way to start writing applications – If you don't know how to start, there are a set of samples that people can follow to see what is a typical code that someone would write when developing an application like this, but otherwise it's mostly about learning, about the concepts in the API especially if you are the event stream API, that would be the basis once you start with, because again those are concepts that exists in other systems in a similar fashion and should be easier for people to start with. Then going to the other APIs and learn about advanced features.

[00:44:12] JM: Is the availability of the data that I've written to Pravega, is it available at an in-memory latency? I know that the tier 1 layers in Apache BookKeeper. Is BookKeeper an in-memory system?

[00:44:31] FJ: No. No. It's not in-memory. No. BookKeeper writes synchronously to a journal. It flushes to the journal because it acknowledges back to the client. It also has the notion of a ledger device that stores data.

If you think about how explain tiering in Pravega, it's a bit similar in a BookKeeper service, which is called a bookie. You write your journal first and you make sure you flush it before you respond and then you asynchronously write to another device so that it can read from. The journal is primarily used for recover. That's how a bookie works. It's durable. It's not in-memory.

[00:45:18] JM: Okay. Does the usage here differ strongly from Kafka in that sense? Because people use Kafka as a pub/sub system for rapidly transferring data from one place to another. It sounds like the usage of Pravega is really more for this storage application.

[00:45:39] FJ: That sounds mostly right. I would say that my understanding of Kafka is that there are applications which are interested in storing data longer term, not necessarily months

or years as I was proposing, but people do want to store data long-term. This rapid exchange of messages, sure, there are some use cases that target that, but I wouldn't say that is only that.

[00:46:07] JM: The format of data that I'm writing to Pravega – In Kafka, you're writing these things called topics. In Pravega, how does the schema compare to that of Kafka? Are you also – Is it topic-based?

[00:46:26] FJ: It's a stream. We call it a stream. We call it a stream and you can specify – There are a few things you can configure. You can specify parallelism, in which case you have parallel segments that you're writing to. That's similar to the topic partitions in Kafka. One big difference is that you can configure dynamic scaling for a Pravega stream. In that case, the parallelism of the stream may change overtime according to the incoming load.

[00:47:02] JM: Tell me more about how the partitioning system works. If I write the stream, that's fine as long as its big enough to fit on a single machine, but eventually it's going to get so big that I'm going to need to partition it. What's the partitioning strategy for a really big stream of data in Pravega?

[00:47:25] FJ: Right. Recall that the stream data ends up in long-term storage, which is a system that can accommodate a lot of data. It's supposed to be elastic. The systems we use for the second tier, they can grow in capacity. So in principal, you should be able to store as much as data as you like or at least you're willing to pay for with respect to storage capacity. There is no limitation from a single server. It's not like a topic partition broken in Kafka where you're limited by that capacity of the broker.

It is also important to note, we only keep data in BookKeeper temporarily. After we have flushed, we have written to long-term storage, then we can truncate that data into BookKeeper. Even if the stream has a lot of data, you have been writing for very long, we don't need to accommodate all the data of the stream of even of a single segment of the stream in a single bookie, right? You only have one, say, portion of the segment in a Bookie for that given segment.

[00:48:34] JM: All right. Well, as we begin to wind down, I'd love to know a little bit about the engineering on the Pravega team. You get a pretty big team. What's the hard problems that you're focused on in Pravega right now?

[00:48:50] FJ: Hard problems we are focused in Pravega right now? We're looking to adding features, say, to support IoT applications and AI. Those areas that we're actively looking to. We are also looking to providing better support for beyond edge and IoT, just geo-distribution in general. That's another area that we explore.

Otherwise, we're looking how to make Pravega better or additional features we can add to improve the situation for AI and ML. There is a lot of interest around machine learning of course. So we want to be a better system for frameworks that support those applications. That's another area that we're looking to. There is a quite a number of exciting things that we're looking at.

[00:49:43] JM: What would that look like if you wanted to a build a system that was better for machine learning applications?

[00:49:50] FJ: One thing, a lot of models that you train typically need a lot of data. The fact to that, we store an amount of data or able to sort an amount of data per stream and you can have it stored in a single system. You can process the data coming out of a single source which could be a Pravega stream. That would be one way of looking at it.

[00:50:16] JM: Okay. Any other developments in the Pravega project that you'd like to discuss?

[00:50:24] FJ: With respect to open source, we are hosted by GitHub. We are looking to two things. One is growing our community, or interested in being a community-driven project. Today, it's one company really working it, but we want to have more companies involved. We are definitely after more collaboration and more partners to work with.

We're also looking to becoming incubating in an existing foundation. That could be – A have a long history with ASF, the Apache Software Foundation, but there are other foundations out there like the Linux Foundation is also very reputable. So those are our options that would be

interesting for us to end up incubating. It's possible that in the near future we'll move out of GitHub and incubate in one of these other foundations.

[00:51:18] JM: Okay. Well, Flavio, thanks a lot for coming on the show. It's been a pleasure talking to you.

[00:51:22] FJ: Yeah. It was a pleasure for me too. Thank you, Jeff.

[END OF INTERVIEW]

[00:51:34] 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]