EPISODE 602

[INTRODUCTION]

[0:00:00.3] JM: A database stores data to an underlying section of storage. If you are an application developer, you might think of your persistent storage system as being the database itself. But at a lower level, that database is writing to block storage, or file storage, or object storage. A container orchestration system manages application containers.

If you want to run WordPress, a blogging platform in a container in Kubernetes, that means you also need a database to store your blog posts in a persistent way. To run a database, you need to have an underlying storage medium, which could be a disk that is on your on prem data center. It could also be block storage on disk at a cloud provider.

Kubernetes is not the only container orchestrator. There's also Cloud Foundry, there's Mesos, there's Docker Swarm, several others. Each of these container orchestrator needs to be able to run a variety of persistent workloads such as a MySQL database or a Kafka cluster. Each of these persistent workloads needs to be able to use different types of backing storage.

With the range of container orchestrators and the range of backing storage types, you get a problem. Every storage type would have to write custom code to connect to each container orchestrator. It's an end-to-end problem. The solution to this is the CSI, the container storage interface. The CSI is a common interface layer between the container orchestrator and the backing storage system.

In today's episode, Jie Yu from Mesosphere describes the motivation for the CSI and gives an overview for its design principles. There are some great lessons here for anyone working with containers or distributed systems in general, and if you're a little bit confused about what the CSI is right now, don't worry, we get into it in this episode. We explain it in great detail.

Before we get to today's episode, I want to just announce that we're looking for writers for Software Engineering Daily. So if you're interested in writing, send me an email to write@softwareengineeringdaily.com . We want to bring in some new voices. We want to deliver

high quality content about software that will stand the test of time, and the container orchestration details of Kubernetes and the container storage interface. This is a perfect example of something that has not been written about much relative to how interesting a topic it is. So if you have something niche, something interesting, something technical that you want to write about, go to softwareengineeringdaily.com/write. Find out more. I'd love to hear from you. We're looking for part-time and full-time software journalists and also volunteer contributors who just want to write about software engineering. We want to explain technical concepts. We want to tell the untold stories of the software world, and we'd love to hear from you. So send me an email or go to softwareengineeringdaily.com/write.

[SPONSOR MESSAGE]

[0:03:11.0] JM: Citus Data can scale your PostgreS database horizontally. For many of you, your PostgreS database is the heart of your application. You chose PostgreS because you trust it. After all, PostgreS is battle tested, trustworthy database software, but are you spending more and more time dealing with scalability issues? Citus distributes your data and your queries across multiple nodes. Are your queries getting slow? Citus can parallelize your SQL queries across multiple nodes dramatically speeding them up and giving you much lower latency.

Are you worried about hitting the limits of single node PostgreS and not being able to grow your app or having to spend your time on database infrastructure instead of creating new features for you application? Available as open source as a database as a service and as enterprise software, Citus makes it simple to shard PostgreS. Go to citusdata.com/sedaily to learn more about how Citus transforms PostgreS into a distributed database. That's citusdata.com/sedaily. Citusdata.com/sedaily.

Get back the time that you're spending on database operations. Companies like Algolia, Prosperworks and Cisco are all using Citus so they no longer have to worry about scaling their database. Try it yourself at citusdata.com/sedaily. That's citusdata.com/sedaily. Thank you to Citus Data for being a sponsor of Software Engineering Daily.

[INTERVIEW]

[0:04:55.8] JM: Jie Yu is an engineer with Mesosphere and he works on the container storage interface. Jie, welcome to Software Engineering Daily.

[0:05:02.9] JY: Thanks. Thanks, Jeff.

[0:05:04.4] JM: I want to talk about storage on container orchestration systems, connecting these two pieces together. I think we should start with a simple example. So if I'm running a container orchestration system, it's probably doing a variety of things. It's helping me run my different applications. One simple application that requires storage is WordPress. WordPress is a blogging platform with a database involved. I need to be able to read and write to that database. What do I need out of my underlying storage system if I want to be able to run WordPress on a container orchestrator, like Kubernetes or Cloud Foundry?

[0:05:46.1] JY: Yeah. Typically, for those applications, you have the stateful part and the stateless part. By stateless part, I mean the application like from the web server and the application business logic that has nothing to do with any state, and the stateful part usually means you have a database or some storage system [inaudible 0:06:04.6] that it can store your state because most of application do require some state. So in this particular case, for WordPress, the stateless part is probably like a web server and the stateful part is probably like a database like a MySQL. Basically you're asking like how do I run MySQL on a container orchestration system like Kubernetes.

Typically, for those databases, they need a file system they can write their data to or [inaudible 0:06:30.5] device they can write their data to. So the underlying container orchestration system needs to provide primitive allowing a database application to write those data to.

[0:06:40.8] JM: So many people think of their database as their backend, but a database is in some ways an application. It's an application that is backed by a more primitive storage element. Help to clarify this – If I have a MySQL database that WordPress is running on top of, what are the different underlying storage mediums that could be underlying that database?

[0:07:09.3] JY: Typically, the database use operating system APIs to talk to the storage systems. So that API is POSIX API, like read, write, those fsync, those kind of storage POSIX

APIs, and operating systems like Linux usually provide different device drivers in the file system APIs to allow those application like MySQL to write their data to. Underneath is the device driver that actually back those file systems cause, like read and write and you can't have different type of device drivers that back those file system API calls, and a device driver can – It is basically a very vendor specific thing. For example, if you have a spindle disk, you have some special device driver for that, and if you have something like EBS or some remote storage [inaudible 0:08:00.4] you have a special device driver for that.

[0:08:03.8] JM: These remote storage backing systems, like Amazon Elastic Block Storage, or Google Persistent Disk, do these necessarily exist on the same physical machine or in the same physical data center as your compute node that the container orchestration system is running on?

[0:08:29.3] JY: Not necessarily, especially for EBS and gcePD, they're remote. They're not at all called to the node and I don't know if they're in a same datacenter or not. There are some restriction on EBS, for example, that it can only access in EBS volume the same zone as the volume.

[0:08:45.9] JM: To clarify, you could have a WordPress blogging platform with a MySQL database that underlies that WordPress instance and the MySQL database application would be running on your Kubernetes node as essentially an application that's running there, the database application, and the backing storage, the Amazon EBS, for example, might be in a different data center. So there might be a network connection that your overall system needs to go over in order to complete a write to your database. Is that correct?

[0:09:27.1] JY: Yes, that's correct.

[0:09:28.8] JM: Okay. So the reason I layout this example is just to give a perspective that there is a lot of complexity and distribution of systems in how the backing storage systems can be interfacing with your container orchestration system. So in the cloud native ecosystem, we have these container orchestrators. We have Kubernetes, Cloud Foundry, Mesos, and then we have this variety of storage vendors. We have Amazon EBS, like we mentioned, Google Persistent Disk, NetApp. There are some other legacy storage vendors. How have the variety of container

orchestrators and the variety of storage types, how have these communicated in the past? Because we've had Cloud Foundry for a while. We've had Mesos for a while at this point. We've had Kubernetes for several years. In these years leading up to the present, how have the container orchestrators and the storage systems communicated?

[0:10:27.2] JY: Typically, before CSI was introduced, I think each different container orchestration system like Kubernetes, Cloud Foundry, Mesos, they all have their own interface internally that the vendor has to implement so that the CO, container orchestration system, can talk to those vendor during the lifecycle of volume. For example, Kubernetes, they have flex volume and also the in-tree volume driver so that as a storage vendor, like I'm a NetApp, I can either write a flex volume base implementation to connect to Kubernetes. I can write an in-tree volume driver for that. So that's for Kubernetes.

For Docker, Mesos, Cloud Foundry, actually, all these three are previously using this interface called Docker Volume Driver Interface and called DVDI. So that's an interface that's kind of internal to Docker, but since Docker is so popular, then those two other container orchestration system decide to use that to talk to the underlying storage vendor through that interface.

[0:11:22.4] JM: So I have a container on Kubernetes, for example. I want to be able to write data from my application container to a persistent storage type. There are many different storage types that I could be writing data to. How does the container know how to connect to all these different storage types?

[0:11:43.9] JY: So from user's perspective, these are the details that's not exposed to the user. If you're a Kubernetes user that you want to use MySQL and want to run MySQL on top of Kubernetes, what you should care about is not underline which vendor you pick all these kind of stuff. You only pick which storage class you need, and Kubernetes has this internal mapping from each storage class to an actual vendor specific parameters and configuration for the storage system. Storage class is like basically like streams, like fast, medium, slow, just a name for a class of storage, and Kubernetes internally map that to a bunch of parameters, and then Kubernetes internally will actually on top to the corresponding vendor through that interface either in-tree or flex volume, now this CSI is being introduced, so there's another way to talk to those vendors through the CSI interface right now.

Through that interface, Kubernetes will drive from the storage specific operation, like creating the volume, or attach the volume, or mount the volume inside this.

[0:12:42.8] JM: Define that term volume. What is a Kubernetes volume?

[0:12:48.0] JY: You can think of volume is a file system that's mounted somewhere in the container that the application can write their data to using the POSIX API I mentioned earlier, like the read, write, the Linux systems calls. The data will be actually persisted as long as the volume object exists.

[0:13:06.3] JM: That persistence, the mechanism by which data is persisted will actually depend on what the storage medium that is backing the volume is, correct?

[0:13:15.9] JY: Right.

[0:13:16.6] JM: Now, does the idea of a volume exist in other container orchestrators talking about Docker Swarm or Cloud Foundry?

[0:13:25.9] JY: Yeah. Docker has Docker volume, which is essentially similar to Kubernetes volume. I think the semantics are very similar because all of them are a file system that's mounted somewhere in the container that you can talk to using POSIX APIs. Yeah, Docker has Docker volume. Cloud Foundry, I'm not so familiar with. Mesos, you have this concept [inaudible 0:13:47.7], which is exactly the same semantics as Kubernetes volume.

[0:13:52.1] JM: Container orchestrators have historically exposed a pluggable storage interface and all of the storage providers have had to adapt to those unique interfaces. Every container orchestrator has their own – Historically has had their own storage interface and all the providers have had to adapt to whatever that interface is. Given an instance, like Amazon EBS would have to Cloud Foundry, and then they would also have to adapt to Kubernetes. So they would have to make multiple plug and play systems for different interfaces. Why is that problematic?

[0:14:31.5] JY: So I think there is a survey. I remember there is a survey a long time ago that the CNCF guys did, like basically for example for just EBS, there are like five plugins out there that they built to adapt to different container orchestration systems. That's just for EBS. For other vendors it's probably the same thing. So it's very painful for vendor, because they're building some software and then they have to adapt to every single container orchestration system, and it is a moving target, because there might be new container orchestration system being introduced. So it's really painful to maintain all of these.

Now that you have this problem, once you build in electric appliances, you have to adapt to different type of electronic outlet standards, which is very painful because when you travel, you have to bring those adapters. It's most painful for vendor, eventually painful for customers and also operators because they need to operate all these vendors and they have to find the right interface to use when they deploy different container orchestration systems.

[0:15:33.9] JM: I think we've outlined the problem here. You've got different container orchestration systems, Cloud Foundry, Mesos, Kubernetes, Docker Swarm. There's several others, but that's just four already. Then you've got different storage providers. You've got Amazon EBS, you got Google Cloud Persistent Disk. You've got NetApp. You have PortWorks. You have all these different storage systems. If there was not some common way for container orchestration systems and storage systems to communicate, you have an end-by-end problem. If you had four container orchestrators and four storage systems, you would have to have 16 interfaces between them.

So the CSI, the container storage interface is an opportunity to connect those two classes of systems, container orchestration systems and storage systems in a unified fashion. Tell me some of the high level design principles behind the CSI.

[0:16:35.7] JY: Yeah. I think the project has started when – I mean, I think we saw a bunch of successful example, previous examples especially now in the storage area, in the now working area initially that the container network interface is a good example of such an interface that bring the container orchestration systems and network vendor together and that turns out to be very successful, and that's the reason we kind of started the CSI project and we have a bunch of conversation with folks in different container orchestration systems. And the first thing we do

actually initially was try to look into existing interfaces that people have already. For example, Docker volume driver interfaces and the flex volume interface, or even the in-tree driver in Kubernetes and we found a bunch of problem there. I mean, there's a bunch of problem [inaudible 0:17:22.7] those problem right now, which I think some of the interface is CLI-based. For example, flex volume. By CLI-based, I mean that the container orchestration system when there's a volume to be attached, you will invoke a CLI binary to actually attach that volume and the storage vendor will provide an implantation of that binary.

But the problem for that is it's really hard to maintain CLI dependencies and think about deploy all the dependency of your CLI on a single box. It's really painful. I used to work at Twitter and I know the tool that we use at the time to deploy such dependency is like puppet, which is super slow and also error-prone and it's very hard to deploy those kind of stuff. That's what containers are meant for. I think some of the interface as CLI-based is not ideal. Some of the interface like Docker Volume Driver interface is problematic because the lack of the item potency. What I mean is in a distributed system, when you talk to some – When a system A talk to system B and the interface within system A, system B if it's not item potent. What that means is you issue two – The same request twice to the same system as you resolve the same thing, but if that's not the case, it's very hard to make it correct especially in the failure scenarios.

Actually, there was a blog post from Stripe engineer at the time writing about in a distributed system you have to design your API always to be item potent. DVDI, Docker volume driver interface, the interface is not item potent, make it really hard to deal with those failure scenarios. Some of the interface that we looked at, for example, the Kubernetes in-tree volume driver, I think at a time they kind of want to get rid of that because I think it create such a burden for especially during the release cycle where everyone wants to jam into – Well, want to have some patch into the Kubernetes core, and it's very hard to core in the release.

Also like after the Kubernetes has been released, whenever there's a bug in a driver, you have to wait for the next Kubernetes release, which is pretty long. It's not ideal for Kubernetes community and also not ideal for the storage vendor and they want to fix that.

Basically, these are the problems we solved from the existing interface and we don't see – There's no solution right now at a time that we can fix all the issues. So we decide to start this

project with the design principle to solve all these issues that we mentioned earlier, like it should not be a CLI-based interface because the dependency management is going to be hard. The API should be item potent so that it's easier to handle those failure scenarios and it should probably not be in-tree interface. It has to be out of tree so that the release cycle can be controlled.

[SPONSOR MESSAGE]

[0:20:08.6] JM: Nobody becomes a develop to solve bugs. We like to develop software because we like to be creative. We like to build new things. But debugging is an unavoidable part of most developer's lives. So you might as well do it as best as you can. You might as well debug as efficiently as you can, and now you can drastically cut the time that it takes you to debug.

Rookout rapid production debugging allows developers to track down issues in production without any additional coding, any redeployment, you don't have to restart your app. Classic debuggers can be difficult to set up, and with a debugger, you often aren't testing the code in a production environment. You're testing it on your own machine or in a staging server.

Rookout lets you debug issues as they are occurring in production. Rookout is modern debugging. You can insert Rookout non-breaking breakpoints to immediately collect any piece of data from your live code and pipeline it anywhere. Even if you never thought about it before or you didn't create instrumentation to collect it, you can insert these non-breaking breakpoints on-the-fly.

Go to rookout.com/sedaily to start a free trial and see how Rookout works. See how much debugging time you can save with this futuristic debugging tool. Rookout integrates with modern tools, like Slack, Datadog, Sentry and New Relic.

Try the debugger of the future, try Rookout at rookout.com/sedaily. That's R-O-O-K-O-U-T.com/sedaily. Thanks to Rookout for being a new sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:22:12.6] JM: We'll go through some of those topics of discussion. Let's start with the in-tree. S in-tree refers to the tree of the source code. So an in-tree solution would mean that you would have driver code for all of these different storage systems in the source code for your container orchestrator. So the container orchestrator source code would have to contain specific code for servicing storage requests to EBS and other source code for managing Google persistent disk, another source code for PortWorks, and that's problematic because every time a storage system would come out, you would have to add in blocks of code in Kubernetes itself that would say, "Yeah, new storage system came out, like google magical store, and we need to be compliant with that." Instead you go with the out of tree solution. Give a little more color on in-tree versus out of tree solutions.

[0:23:12.5] JY: Yeah. In-tree as you mentioned, the source code of the vendor code is actually part of the container orchestration system. Typically, that means that those color will be running in the same Linux process, OS process as the CO, container orchestration system. But the problem for that is, first of all, the release cycle. As I mentioned, the release cycle is tied together, which is not ideal. The second thing about the issue is the security, for example, because your code is actually running in a same OS process as the CO. You have the same privilege as the CO, but the CO itself is not actually written by CO. It's written by storage vendor and there needs to be a trust between CO and those storage vendors CO, but it's really hard to achieve that. So then that calls some of the security issues.

By out of tree is the storage plugin will actually be running in a different – First of all, out of tree means the CO itself for the storage vendor is actually in a different repo than the container orchestration system's code. So usually if that's the case, it typically means that the CO itself will be running a separate OS process as the CO, because I think if they are running in the same OS process, the only way you can do that is through dynamic linking, which is not portable across multiple languages. [inaudible 0:24:26.4] we want to achieve is like because Kubernetes is written in Go, Mesos is written in C++ and Docker is written in GO. So we want to deal with all different languages. We don't want to tie to a particular language. Dynamic linking is not an option for us.

So what we mean out of tree is always mean like it's out of process too. So the plugin itself is running in a separate process and container orchestration system will talk to those plugin process through some APIs to make those modern cycle, volume lifecycle calls, like attach volume, create volume.

[0:24:58.2] JM: There's a question here, where the code should be. The code for the container storage interface, you put it in the control plane only, not the data plane. Describe this in more detail. What is the control plane? What does that mean specifically and why is the CSI code localized to the control plane versus the data plane?

[0:25:22.5] JY: Yeah. So control plane and data plane, as far as I know, it's kind of borrow from the networking terminology, because that used to be the term in a networking area, where a control plane means it control where the package go, but didn't dictate how the package flow through on the network. For example, like routing, like when you have a package, you decide where the package should go. That's control plane operations. But how the package actually flow into a different host or different router, that's data plane issues. It's not control plane.

So map that to storage, control plane means like how those volume are actually being set up and connected to the CO. In data plane, in storage, it actually means like how these data actually flows. For example, through [inaudible 0:26:05.4] protocol flow through like fiber channel or like some TCP/IP based protocol for those storage actual data to flow from one host to a different host from container orchestration systems through the storage vendor. We don't try to define that protocol. We only want to define the protocol where specify how the volume should be connected. How do we setup the volume? How do we mount the volume? We don't care about how the volume data will go and how that should go.

[0:26:31.9] JM: It's been really important for the CSI to be vendor neutral, and this is part of Kubernetes drawing on lessons from past open source systems where there were vendor wars over control over certain areas of the storage systems. I'm not sure exactly which open source systems, and I probably shouldn't even name names, but there have been open source projects in the past where vendors have gotten involved to much in specifications that should have been non-vendor specific. Because of that, the whole ecosystem can get contaminated with these tribalistic business interests.

Vendor neutrality, how do you – I think, first of all, why wouldn't be this vendor neutral? What is the difficulty in making your container storage interface vendor neutral? How would you potentially favor a vendor and how do you avoid that?

[0:27:37.7] JY: Well, I think by vendor neutral, I think basically for the specification itself, no vendor can actually dictate the direction of the spec. As you said, that usually means like if we have some vendor that have a control on the specification, they will put some of the vendor specific features or proprietary features into the spec, because vendors always competing with each other to have some proprietary feature. They want to be shiny in the CO systems, and if we let one vendor control that thing, then most likely that the spec will favor one particular vendor and will kind of be causing issue for other vendors.

What we did actually trying to avoid the interesting way for this CSI project specifically is we say that only container orchestration system, community members, representative can be approver for that spec so that if folks aren't CO's perspective and don't allow any storage vendor to be an approver for this spec, so that we avoid this issue, like having one vendor control the spec. Things, like container orchestration system from their perspective, it wants to be vendor neutral, because it wants to work with any vendor, any storage vendor. I think the goal is aligned. So I think that model will make it easier for us to make the spec vendor neutral and that's actually what we did.

[0:28:54.7] JM: There's a number of design questions around building this container storage interface's distributed systems design questions and software architecture design questions. The CSI, it could work through a CLI, command line interface or it could work through a long-running service. So you could have the storage vendor have to deploy a binary on a host in order to be executed to a CLI. In that situation, the container orchestrator would invoke the binary and that would allow the container orchestrator to connect to the storage system. Then alternatively, the service model, the container orchestration vendor would deploy a service on the host, and then the orchestrator would make requests to the service and then the service would broker the connection between the container orchestrator and the storage system. Why is this an important question? How did these two approaches to connecting a storage system to a container orchestrator. How do these two approaches contrast?

[0:30:06.5] JY: Yeah. I think I kind of briefly mentioned that the design goal that we want to achieve and we're looking to existing interface at a time and most of the interface actually is CLI based. For example, the flex volume is CLI based. On the CNI, the container network interface is a CLI based interface and we have this debate whether we should go with service versus CLI. I think that the main complain that people have with a CLI based interface is it's super hard to deploy compared to a service which you can just use a container. That's the reason container orchestrations exist, deploy containers. It's much easier to do that with one service than deploy CLIs.

I think the other problem is, I think, if a CLI based binary, if you call the CLI interface, usually the CLI binary require will access to do things, and it's not very safe to allow an arbitrary binary that's written by the vendor to do that. By running a service, you can use many of these container primitive, like Linux capabilities or [inaudible 0:31:07.9] user and give grants and additional capabilities to kind of restrict that access.

Also, I think CLI, like the dependency management, these are kind of related to the deploy issue where the dependency of the CLI is hard to deploy, because there's not really an easy way to deploy those binaries. The only way I can think of right now is using those like [inaudible 0:31:27.2], Ansible, those kind of stuff to deploy their CLI binary dependencies, which is not easy to upgrade or maintain.

Also, I think specifically for storage compared to networking. The storage has one unique requirement, which is there are some search and file system called fuse where the file system itself is running in the user land, not in the kernel space. For field space backends, things like S3FS, those kind of fuse based backend, it will require a long running process anyway. Basically, the long running process is the one that's serving those file system API calls. The long running service – If the long running service is down, the file system is down.

For storage specifically, we have this requirement that some of the backend require a long running service anyway. I think in that case, a service makes more sense because then you can just jam all these dependency into one single container including the plugin interface as well as these long running services for fuse backend. That makes the decision much easier – I mean, I

think that's a pretty natural decision that would go with service, because all these kind of issues with CLI and also there are some special requirement for storage to have a long running service running.

[0:32:33.2] JM: You talked earlier about the item potency question. So all API calls between the container orchestrator and the storage system, the storage system that you're interfacing with. All these calls should be item potent, and item potency means that an operation could be applied multiple times without changing the result that the initial application of that operation had. For example, if the container orchestrator issues a call to the storage system that says, "Hey, I want to provision the amount of space that I need for a volume on my container orchestrator, or I want to be able to have this volume abstraction so I can interface with it and write database entries to it.

If you made that call to the storage system, you could imagine all kinds of networking failures that would result in a block of storage being allocated on the storage system, and then maybe if the call fails and that storage block got allocated, but it never got assigned within the container orchestrator to a specific volume. Then the container orchestrator might retry and then the storage system would spin up another blob of storage, and then the first blog of storage would be orphaned. I mean, that kind of thing would be problematic. Describe some of the difficulties around item potency in the interaction between a container orchestrator and the storage system.

[0:34:05.7] JY: Yeah. The example you give is exactly right, the exact problem we face when we deal with previous interface, like Docker volume driver interface. For example, the create volume call in a DVDI interface is not item potent. You don't specify the idea of the volume into the create volume call. Instead it will return an ID to you and if the response you – The CO didn't receive the response, then that volume created by the backend will be leaked [inaudible 0:34:30.7] have a handle to that volume, and that's a big problem for a full storage system and that's [inaudible 0:34:35.4] we're trying to kind of fix that by requesting that, "Oh, the create volume call should be item potent." What that means is essentially you have to specify a name or an ID of the volume where you make the call and so that the CO has a reference to that volume handle, like name or the ID and so that even if you don't receive the response, you can try that call again and then we dictate the storage providers makes sure that if the same volume

ID is used, it should result in the same result. It will give you – It will return a success eventually to the CO, and CO will receive the response and processing the rest.

I think these are just based on the experience that we have when we're building such a system. I think another example for that is AWS EBS API. Itself is not item potent, because when you're actually creating the EBS volume, the call don't allow you to adding tags atomically. It can only add a tag once the volume is created. So that's problematic, because once you create a volume, if you don't receive the handle, that volume will be leaked. The same issue that I described earlier. Then we face a lot of issue with EBS due to that, and that makes us think that item potency is really important, otherwise it will be so much painful to recover those create volumes. So that's the reason we put that as a top priority for the CSI spec when we first discussed that to trying to solve those real issue that we saw in production.

[0:35:58.8] JM: Can you solve item potency at the specification level or do the vendors that are writing to the container storage interface that are writing their interface, like if I'm Amazon, I'm writing in my EBS connector, my CSO compliant, my container storage interface compliant connector to connect to Kubernetes, to connect to any container orchestration system. This would connect to Kubernetes. It would connect to Cloud Foundry. Do I have to specify the strategy for my item potency?

[0:36:32.5] JY: Well, I think the spec dictate that the implantation should make the call item potent. In the interface, for example, create volume. In the interface, there's a field called name. So the CO will actually specify the name, the volume, where it should create volume call. So it's plugins responsibility to make sure that call is item potent. What that means, if the CO issue the same create volume with the same name, only one volume should be created. It's plugins responsibility to satisfy that requirement from the spec. I know that some of the source system might not be able to achieve that using their existing APIs. So that's their job to fixing their API to make the life of the CO much easier. Otherwise, just like there's no way to fix that failure recover issue in the case of like response get dropped. So that's a historically issue that we saw, and I think that's a way to drive those vendors to fix their API to make their APIs item potent so that they can satisfy the spec so that we don't have this issue. Otherwise, this issue will never get fixed.

[0:37:37.3] JM: So this is a serious issue.

[0:37:38.7] JY: Right, in production.

[0:37:40.7] JM: In production. So is the consequence of this that you just get like wasted storage space, this orphaned storage space problem or are there more severe consequences to not having item potency here?

[0:37:53.5] JY: I think it's mainly the leaked volume. I think that's for the create volume call. For other calls, it just cause issues with CO. For example, when you do a controller publish, which is essentially just attach a volume to a given box. If that response you don't receive the response for that call, the CO don't know what to do the next, because the volume might be already attached to a volume or might not be attached to the volume and CO might need to – Without the CSI, the CO might need to use some different mechanism to figure out whether the volume is attached or not. It just creates so much pain for CO to deal with that kind of logic.

The interface, trying to solve that problem by define exact semantics and say, "Hey, it has to be item potent." Simplify the life for CO and also help to kind of alleviate those failure recurring problems. The storage vendor – It creates some burden on a storage vendor, of course, but I think that's the right direction. I think the blog post that the Stripe engineer wrote at the time basically saying that any distributed system, if you want to design a robust and predictable API, you have to make the API item potent. I think that should be the first design goal when you start to work on a distributed system.

[0:39:01.6] JM: I'll certainly put that link in the show notes. I want to check that out myself. It actually doesn't sound that hard for a storage vendor to implement this, to fix whatever issues they would have, because if the spec is that the container orchestrator declares a name and an ID associated with the backing storage block or a quantity of storage that they're requesting, and then they communicate that to the backing storage system, and then the backing storage system checks if does something already exist on my side with that name and ID. If so, then maybe I just connect that to the container orchestrator and things are fine. If it doesn't, then I instantiate it. It shouldn't be that complicated, right?

[0:39:50.4] JY: Yeah. Actually, let me give you one example, which is EBS. At the time we looked at EBS API. It's very hard to make that call item potent. The reason for that is AWS don't give you an atomic way to say, "Hey, I want to create a volume and also attach some of the tags to the volume." I haven't checked the recent API, but at the time we checked the API. It's like that.

What that means is you have to create the volume first, get the volume ID and then attach some tags to the volume, which is essentially the name that's specified by the CO. But it's not atomic. Basically it means that if the plugin crashes or the CO crashes in the middle, once the volume is created, but the tag has not been attached to the volume, that volume will still be orphaned. It's about the design.

I mean, [inaudible 0:40:32.6] probably don't have that issue, because they allow an atomic attach of tag to a given volume. So that can make the call item potent pretty easily just to say, "Hey, attach this given name to the volume." It can still return me a volume ID. That's fine, but the volume will have a tag so that I can check whether the tag exists for that volume if something fails before. So that's easier. But for some vendor, it's not that easy using their existing API. I think that's kind of a good thing. I think the spec kind of force them to fix that issue, because I think that has been the issue for a long time and people knows about that issue, but no one is trying to fix that.

Integrate with your choice of container registry, including Azure container registry. Also, quickly and efficiently scale to maximize your resource utilization without having to take your applications offline. Isolate your application from infrastructure failures and transparently scale the underlying infrastructure to meet growing demands, all while increasing the security, reliability and availability of critical business workloads with Azure.

To learn more about Azure Container Service and other Azure services as well as receive a free e-book by Brendan Burns, go to aka.ms/sedaily. Brendan Burns is the creator of Kubernetes and his e-book is about some of the distributed systems design lessons that he has learned building Kubernetes. That e-book is available at aka.ms/sedaily.

[INTERVIEW CONTINUED]

[0:42:50.0] JM: Continuing the conversation of this being a distributed systems specification that we're trying to design here. The APIs between the container orchestrator and the storage system could be synchronous or they could be asynchronous. If they were synchronous, then a given API call would be – You would have guarantees that it would be executed atomically. Everything within the system call would execute before the system proceeded. Asynchronously would mean that you would initiate a request from the container orchestrator to the storage system and then it would be non-blocking and the container orchestrator will continue doing work and then eventually we get a callback from the storage system and finish up whatever other kinds of work you have to do around the API you've made. What are the tradeoffs between synchronous and asynchronous APIs between the container orchestrator and the storage system?

[0:43:46.4] JY: Yeah, I think the main reason want asynchronous, especially the storage API is because some of the storage operation is super long. For example, to attach a volume or detach a volume might take minutes or like tens of – Like 30 minutes. I've seen cases where like a detach take 30 minutes. So since it's super long and when an operation is long, the natural design question is whether this API should be async so that I have a callback. In the meantime you can start to process some other operations. But the tradeoff of an async API is that it's significantly more complex than a synchronous API, because then you have to have some sort of ID for your operation, and when you receive a callback, you have to correlate that

response to a previously pending operation. So it creates so much complexity into the COs code.

Also, I think the async itself – The reason people want async is because they think that it solved the long running operation problem by using an asynchronous operation, but it really doesn't, because at the end of the day, the CO has to time out anyway, because if the CO didn't receive a response or callback after like 30 minutes, it has to time out just to be defensive. What if the storage system is completely down or there's no recovery, there's no operator coming to fix the problem? So the CO needs the defensive anyway to deal with those kind of scenarios. Async really doesn't have in that scenario.

I think the key here is trying to make the call item potent so that CO can just safely retry with the same operation and expect the same result. If it doesn't receive the response, it would retry again until the timeout happens.

The plugging implementation can still be async. Just the interface between CO and the storage vendor has to be synchronous for the sake of simplicity. Plugging can choose their implementation. It can be async for sure, and I now many people choose to be async for long running operations. That's totally fine.

[0:45:39.6] JM: Let's give an example here. I want to create a database on top of a volume on a container orchestration system. Now that we've given so much detail into what the container storage interface does, describe what happens. If I want to create a database for my WordPress blog, on my container orchestrator, what's going to happen behind the scenes? Maybe you could give one or two examples of different storage systems that would potentially be a good fit for backing this database that I needed to create and how these would be created and connected to my container orchestrator.

[0:46:18.2] JY: Yeah. So let's just EBS as an example and also using Kubernetes. So when you deploy your –

[0:46:25.1] JM: Sorry. Amazon Elastic Block Storage, just for people who don't know.

[0:46:28.6] JY: Right. Okay. I use Kubernetes as an example just to demonstrate how this whole thing works. The operator will actually talk to API server to create your MySQL application, a bunch of pod, and a pod is just a collection of container that's running on the same network name space. When the CO first needs to make a scheduling decision to where to place those pod. Once that scheduling decision has been made, seem the container, the MySQL container, needs a volume to store its data, a persistent volume to store its data, the CO will find out that, say, the MySQL pod needs a volume and the agent component of the CO in this case is a kubelet, will actually try to make that volume available on the node so that MySQL application container can actually write to the volume.

So the CO will actually – At this time, will actually try to attach the volume. Let's assume the volume already exists. I'm going to back to talk about if the volume does not exist, what's the workflow? If the volume already exists, the COs will actually just invoke the CSI interface to attach the volume to a given node that the scheduler pick and mount the volume on a specific location on the node. Then using Docker to launch the container and when the volume for the container is actually that mounted file system on the node already that previously done by the CO by invoking the CSI interface.

So that's kind of the launch path where you have the workload reference to a persistent volume, and assuming the volume already exists. If the volume does not exist – So in Kubernetes, the typical way is the container will – The [inaudible 0:48:07.6] will actually specify a persistent volume claim saying, "I want a 10 gig volume that has storage class full," and a storage class as I mentioned is an indirection from a name to a set of vendor specific parameter for that class or storage, and Kubernetes will actually translate that volume, storage class fast to a bunch of vendor specific parameters and actually call the CSI interface when [inaudible 0:48:34.5] a persistent volume claim and if there's no persistent volume that binds to that claim yet, you're trying to create a new persistent volume that can satisfy the requirement of that claim. At the time, we're trying to call CSI create volume, trying to create a volume, persistent volume by using that CSI interface and the backend will actually provision an actual EBS volume for that persistent volume claim and this will bind to that persistent volume claim. Then the rest will be the same, that the [inaudible 0:49:02.5] will be scheduled by the scheduler on node. The kubelet will actually make the volume available on a node by invoking the CSI interface publish volume and controller publish and node publish.

[0:49:13.3] JM: Once that database is wired up and it's connected to the backing EBS storage, what happens if my database application container, or my container orchestration system. What happens if my database application container dies?

[0:49:30.3] JY: If the application dies, the scheduler will make a decision to where – I mean, basically, the scheduler will try to restart the same application not necessarily on the same node, but potentially on a different node. So if that happens, for example, the scheduler decides to place the application on a different node, then the kubelet will actually try to make sure that the same volume will be accessible from the second node and it will try to do the detach first. For the EBS case, you're trying to detach EBS volume from the previous node first and then attach to the new node, and that all process, if through CSI by calling some specific CSI interface, and unpub controller no publish volume, and then controller publish to a new node and controller node publish to the node.

[0:50:12.3] JM: Right. If you have your database application container and that container is scheduled on a pod and the volume is connected to a specific pod, because each volume in Kubernetes is connected to a specific pod, that volume is connected to your database application container. Your database application container dies. So the database application container is going to get rescheduled on to either that node, in which case it can just reconnect that node and it will probably get scheduled to the same pod or if it gets scheduled to a different pod, then your volume could just get unmounted then reconnected to the other pod on the same node. Alternatively, if there's a different node, as you said, the volume would get disconnected from the previous node and then would get reconnected to that new node. Okay. We walked through that failure scenarios. Is there a failure scenario that would be common in the case of the storage system failing? Your EBS – I don't know much about Amazon EBS, but can that system fail and then does the container orchestrator have to reschedule for a new storage backing system? Does it have to reschedule in that instance?

[0:51:33.4] JY: This is typically – It's a hard problem and usually require operator to intervene. For example, usually, when you discover this problem, usually some metrics of the application self goes wrong. For example, your right time for each transaction, like the time you need to perform a transaction goes extremely high and those metrics trigger some alerts and the

operator will come in and will get paged. First get paged, “Why this metrics goes so high?” and trying to figure out the root cause. There might be some diagnosis information they can collect for each individual vendor to indicate, “Oh, this volume, the disk goes bad. It needs to be rescheduled.” If that happen, the usually like manual intervening is required and the operator will usually like replace the disk. In the traditional world, the operator will just replace the local disk. In the cloud native environment, I think that you can just reschedule – I mean, if the database itself is replicated, then you can just start a new node with a fresh disk and it will start replicate itself.

If the database itself is not replicated and the data goes bad on your disk, I don’t know what you can do. You can try very hard to recover your data from the disk, but this is pretty rare given that, for example, EBS has a replication itself. So the probability that this happens I pretty low. But if that happens, I think there’s nothing you can do. The data is being uncorrupted on the [inaudible 0:52:58.7] and if you don’t have replication, then it’s a bad situation. Anyway.

[0:53:03.2] JM: Let’s zoom out a little bit. Why is this interface important to Mesosphere? You work at Mesosphere. Mesosphere is a business, and obviously it’s based on open source technology, and so open source technology is somewhat important at a core level to Mesosphere. Why is the container storage interface uniquely important such that your full-time job right now, or maybe it’s not your full-time job, but you spend a lot of time on this container storage interface, which is an open source community interface project. Why is this important enough that Mesosphere has allocated resources to it?

[0:53:42.2] JY: Yeah. I think from company’s perspective, from Mesosphere’s perspective, we want to solve customer issues. I think one of the issue that our customer has is storage, because we see a bunch of issue with our previous interface that we use for storage integration, which is DVDI. But as I said earlier that we find out a lot of the issues, real production issues for those because of using that interface, and that’s a reason kind of we start to think and also like talking to other folks from different COs to see if they have the same problem. It turns out that they have the same problem and we are trying to solve the same problem. So that’s the motivation from our company’s perspective, because we have customer that ran into real issues with their existing interfaces and they want to fix that and we also talk to other COs and they have the same issue. So kind of our goal is kind of aligned and great, so we need to figure out a

new interface so that it will benefit everyone, benefit our customers, benefit their customers, benefit storage vendors too. So that's a reason like we spend resources to build this community. We want that to be successful. We want to reduce the burden of storage vendor. We want to make our customer happy because we're using that new interface that solve some of the real production issues for our customers.

[0:54:58.2] JM: So you and I were both at KubeCon in Copenhagen recently, and what I thought – I guess what I took away from a business point of view at KubeCon was that there were a lot of enterprises that were very interested in buying different products from container product vendors. So there're all these vendors in the container space. There's obviously orchestration vendors, like Mesosphere. There are security vendors. There's monitoring and tracing and all these different products, and enterprises are ready to buy this stuff. I don't know how much insight you have into this, but tell me about the buyer these days, the enterprise buyers, the type of customer that Mesosphere is catering to. Maybe they've got a lot of legacy systems. What are they looking for? When they shop around for these different container product provider vendors, what are they looking for?

[0:56:00.6] JY: Yeah. We have customers from different – We are targeting for like fortune 500, maybe fortune 2000 enterprises. The reason they want to buy the software is because they don't have the people resource to build them self and there are like specific features they're looking for that our platform provides. For example, one example is like security, like many of the banking – We have a lot of banking customers that – They have a very strict security policy and they require some security feature that would be on top of the open source solutions.

Some customer, for example, the telco – We have a lot of customer in the telco industry and a lot of the case they're looking for are something like IoT like thing and the reason that they buy Mesosphere is because Mesosphere provide this platform that can run not just stateless applications. It can also run stateful application, things like Cassandra, Kafka, and also it can run analytics workloads like Spark. That kind of give kind of very coherent story, like you have your IoT devices that collect data and that will be ingested into a pub/sub like Kafka, and then you have your database, your key value store Cassandra and you have Spark that can actually subscribe to these pub/system like Kafka and to do either like analytics and also persist those data. It's just like a very nice storyline for them to solve their real problems. I think a lot of

customer buy us because we provide not just stateless solutions, also stateful solution like Cassandra, Kafka, HTFS, these things that the customer need and they don't have resources to build themselves.

[0:57:39.6] JM: All right. Jie, it's been great talking to you. I want to thank you for coming on the show and talking about container storage.

[0:57:44.7] JY: Thank you. Thank you, Jeff. It's nice talking to you too.

[END OF INTERVIEW]

[0:57:50.2] JM: GoCD is a continuous delivery tool created by ThoughtWorks. It's open source and free to use, and GoCD has all the features you need for continuous delivery. Model your deployment pipelines without installing any plug-ins. Use the value stream map to visualize your end-to-end workflow, and if you use Kubernetes, GoCD is a natural fit to add continuous delivery to your project.

With GoCD running on Kubernetes, you define your build workflow and let GoCD provision and scale your infrastructure on-the-fly. GoCD agents use Kubernetes to scale as needed. Check out gocd.org/sedaily and learn about how you can get started. GoCD was built with the learnings of the ThoughtWorks engineering team who have talked about building the product in previous episodes of Software Engineering Daily, and it's great to see the continued progress on GoCD with the new Kubernetes integrations. You can check it out for yourself at gocd.org/sedaily.

Thank you so much to ThoughtWorks for being a longtime sponsor of Software Engineering Daily. We are proud to have ThoughtWorks and GoCD as sponsors of the show.

[END]