Rasmus Bach Krabbe (00:05):

Hi, folks. My name is Rasmus. This is Frederik. We are from City Storage Systems. And we're going to tell you a little bit about our experience running CockroachDB on Kubernetes for the last around three years. So first off, what is City Storage Systems even? So we're a startup in the real estate and food tech space. So we buy dilapidated real estate all over the world, install state-of-the-art kitchens, and we help restaurant entrepreneurs open up delivery only kitchens. So to support this, we also offer a suite of software like the other suite of software for restaurant owners to grow their digital business and integrate with online food ordering platforms. So we use CockroachDB for the bulk of our old TP load. We have other databases like Postgres, but this is the primary database for our most critical workloads. So Fred and I are from the Infrastructure Core Storage team.

(00:57):

We run a largest fleet of CockroachDB clusters, and a lot of those are multi-region. So we're a pretty small team. We focus on delivering storage as a service and not so much on database administration. So why are we running CockroachDB? So like a lot of other folks that have been on the stage for high availability, being able to do graceful rollouts, stuff like that, we run across multiple regions to be able to tolerate the outage of an entire cloud region. We've had a strategic ambition as a company to be multi-cloud from day one, so we wanted a technology that was portable and could run at different cloud providers, and we wanted something that could speed up application development by offering a strongly consistent storage solution across these multiple regions.

(01:43):

So what we're going to go over today is we're going to go over a little bit about the beginning. How did we get off the ground on commodities with CockroachDB, StatefulSets and home releases, getting stuff up and running. Then we're going to tell you about how we started automating things and making our operations more efficient, and then we're going to tell you a little bit about some of the more recent stuff we've been doing and what we're looking at in the future.

Frederik Stenum Mogensen (02:03):

Thank you. So in this first section, I'm going to take you through our journey from the start with how do we actually get CockroachDB up and running on Kubernetes? But first off, why actually Kubernetes? Well, we choose Kubernetes from much the same regions that we choose CockroachDB. We wanted availability, and Kubernetes gives us a lot of tools for restarting stuff that breaks and moving between nodes and killing off [inaudible 00:02:33] without this actually having an impact on the workloads on top. We want portability, and Kubernetes gives us a vendor-agnostic runtime, so we can run Kubernetes on everything and cloud provider, most of the cloud providers have a managed solution already, so we just use that out as a box, and we wanted the development speed of it.

[NEW_PARAGRAPH]And Kubernetes is increasingly becoming the industry standard to run stuff in the clouds, and it gives us the added benefit of actually being available on my local laptop. So I can use my local laptop to run all of my integrations tests going with this optimization. Can I get Cockroach running and does it do the right thing before I push it into station and production? So how to actually get Cockroach up and running on Kubernetes? We started off here. We start off with a very simple helm chart, which has Native Kubernetes resources in it. That's the StatefulSet that sets up the parts, and the parts is what's running the actual Clockworks nodes fine.

(03:29):

We deploy this into three different regions. This gives us Clockwork nodes in each region, and we go in manually and initialize the clusters. We have a coherent Clockwork cluster that we can start using, create users on it, administrating users and actual customer users, and create database and grant permissions, and create the actual client keys and share them with the clients. This all works very nicely, but this has a lot of toil for us as a small team trying to manage a fleet of Clockwork clusters. So there's a lot of manual steps here, and we all know that when there's a lot of manual steps, someone is going to forget one of them or do a typo and the users will have the wrong name forever and stuff like this, so this is one of the big pains in the provisioning part here.

(04:13):

It works. It just has toil. The more interesting thing is when we get to the day two operations of these things, we set up all of our clusters with very small disks because money isn't free and we want to save as much money until we actually need the capacity. And when we start using the clusters and people start using the disks, putting in more and more data, we need to scale up the disk. Unfortunately, the StatefulSet as a native resource in Kubernetes doesn't have this feature. The size of the disk is at immutable field. You can't change the actual size underneath the StatefulSet. So what you do is, or at least what we did is we delete the StatefulSet. We don't need this thing anymore. We remember to put in the flag that goes, "Please let the parts keep on running," because if we don't, then we just delete our entire Clockwork cluster.

(04:58):

That's a good thing in production, and we go in manually and patch seeds of the PCs underneath the actual disks and go, "Please, dear cloud provider, make this one bigger and make this one bigger." And we redeploy the StatefulSet having the right size in the template so that every new node that comes up will actually have the new disk size that we want, and then we roll all the parts to ensure that the StatefulSet actually starts owning the old parts you have. So there's process for this. It works. It has some pitfalls that are interesting, such as, if anyone ever forgets the cascade often flag when deleting the StatefulSet, that just deletes the entire cluster for you. You didn't actually do this with Cockroach. We did lose some Kafka clusters this way in stating [inaudible 00:05:43]. Don't talk about that.

(05:46):

So hope that the parts doesn't fail while the StatefulSet is gone. The StatefulSet owns the parts and orchestrates them. So when the part goes down, StatefulSet goes, "Oh, you need this one. Bring it back up." If you actually delete the StatefulSet, there's no controller anymore to ensure that the part comes back. So when you delete the StatefulSet and you start patching stuff and before you reapply it, if you lose a part, well, then it's gone and you're going to have less and less capacity in your Clockworks cluster and you're going to have unreplicated data and eventually unavailable data. This also requires me as a developer to have way too big permissions in my Clockworks clusters, a community cluster, sorry. I don't want to be able to just delete my clusters in production, but I have to if this is the normal working procedure. And the last thing is that this only works for making disks bigger. This is what we needed at the [inaudible 00:06:34], that's okay. Another day two operation, which is interesting is the scale in and scale out. Here for scale in we have too much capacity in our cluster, but we want to run them more lean. We want to actually utilize our resources. So for manually, for scaling in a Cockroach cluster on Kubernetes, the process will look something like this. Go in, you use the Clockwork CLI and go, "I want to drain that node. Please move away all the data and all the theses," and depending on how big the nodes are, this may take minutes or hours, or days or weeks for some nodes with a lot of data on. And then when the node is decommissioned, you go to the StatefulSet, "Please scale this one down," and then you go on to the next one.

(07:17):

So we all the StatefulSet here, I want to go from three nodes to one node in this specific region. It removes the parts after we have decommissioned them, and the PVC is just left floating around in your company cluster. You have to manually remember to delete those if you want the money back as you're just paying for disks you're not using anymore. Again, this requires me as a developer to actually have the permissions of deleting disks in production from my local laptop, and that will keep me up at night, and it may take many days. For me to scale in a cluster could take two weeks of having to remember every day to going, "Is this done? Yeah, this is done," we move and do the next step.

(07:55):

When scaling up again, we need the capacity back. This is much simpler. Just go to the StatefulSet and go, "I want three nodes instead of one node," and the StatefulSet will bring up the nodes, which is very nice. The problem here is if you did a scaling earlier and you forgot to delete the PVCs, or you were too scared to delete the PVCs or whatever, then the parts that come up will actually be zombie parts because they will already be decommissioned. It'll have the state on this that goes, "I'm not doing anything. I'm not going to add in anything to the running Clockworks cluster." So this brings you in a state where if you don't notice it, you think you have more capacity in the cluster, but you don't actually, you need to fix that.

(08:34):

That's also the interesting part, when we run multi regions of, if you add in a new node in one cluster and it joins the Clockworks cluster, but you haven't solved server [inaudible 00:08:46] in the other clusters, then when the node starts talking to the node in the other multi regions, in the other regions, they can't call back again. So if you have the situation where a node comes up, it becomes leaseholder for some range because we want to share the load to it, and then the courier comes in region B and-

Frederik Stenum Mogensen (09:00):

And then the courier comes in, in region B and it sees, "Oh, the leaseholder is that new guy. I'm going to talk to him to get the data." It's not going to be possible. This gives you an asymmetric network partition and this gives you a lot of pain. We had many outages with asymmetric network partitions before we solved this in our automation.

(09:19):

So the journey so far of running actually Cockroach on top of Kubernetes has been okay, right? We can get clusters up and running on Kubernetes. It's fairly simple. It's fairly stable. The network attached disk we use from the cloud providers, they're fast enough and they're very convenient. They're mainly fast enough because, well, most of our latency is dominated by cross region requests. So this is not a concern for us that it may take an extra millisecond to talk to the disk.

(09:47):

The Cockroach clusters we are running are highly variable and they are multi-regional, and they can handle regional failures without problems as long as we set the [inaudible 00:09:57] correctly on our notes. Service discovery across multiple Kubernetes clusters is not trivial and it needs to be solved in some way. And well, we solved it in a value okay way for this part. We added more automation later that made it much nicer. And lastly, when running in this way, using the native Kubernetes resources such as the StatefulSet, the StatefulSet has a lot of immutable fields, which is kind of a pain because you can't change your Cockroach clusters afterwards. So this makes the day two operations much more tricky.

(10:32):

In the next section here, Rasmus is going to tell you a lot about how we address this toil that comes from this section and how we start in all the automation for running this much more lean and much simpler.

Rasmus Bach Krabbe (10:46):

Yeah, so we started thinking about how can we start to automate more of these operations, remove toil from the team. Running on Kubernetes, one of the predominant patterns for automating stuff like this is to do a Kubernetes operator. So a Kubernetes operator is a pattern where you extend the set of resources available in Kubernetes by creating some custom resource definitions. So this is something domain specific like a CockroachDB cluster or something like that, and you can attach some code via Kubernetes controllers that is going to run when you update or create these kinds of resources. So this gives us declarative management of resources like databases and users, and it gives us the ability to run code to orchestrate some of these complex operations that Frederik was talking about, like no decommissioning generating client certificates and scaling.

(11:39):

So when we started down this path, we were looking at the open source Cockroach labs operator, and sadly, four out of the five limitations that are stated on the operator were kind of deal breakers to us. We very much wanted to be multicloud. The open source operator was not, it did not support multi region. There was no defined path for getting off of Helm Charts and onto the Operator. And we run Istio as our service mesh, and the Operator has not been tested with this. So we decided to write an in-house operator. We started implementing this in the second half of 2021, and we shipped to production before the end of the year. So this was not a huge undertaking. We got off the ground fairly quickly.

(12:23):

So let's look a little bit at what does this look like? So the developer experience within storage ended up being we have a YAML document, something like this. It defines that we want a cluster of some sort. We want a set of nodes. We can have multiple of these with different characteristics. We define the databases, the users, the frequency of backups and all of this stuff in one place. And then we deploy this to Kubernetes.

(12:47):

So what we get inside Kubernetes is a hierarchy of resources, kind of like this. We have the CRDB cluster that defines the logical cluster, and then we have a set of resources around it, the logical configuration of users, databases and settings. And then we have the CRDB node set, which wraps the StatefulSet and allows us to orchestrate some of the operations Frederik talked about in a more fine-grained manner.

(13:13):

So what does scaling look like, for instance, in this world? So when we were adding nodes, one of the pain points was that we needed some way of doing cross Kubernetes cluster service discovery. What we did here was we leveraged the fact that we are running Istio as a global service mesh. So Istio already knows about the endpoints that are coming up in different regions. We just needed a way to propagate the information into Kubernetes, into Cockroach to actually make DNS lookups work. What we did was we scraped the Istio control plane information around which endpoints exist, and then we create the necessary service discovery resources in all regions.

(13:52):

We also added orchestration of the decommissioning process when removing nodes. To do this, we had two options. One of them was we could shell out in the operator and call Cockroach node decommission. We felt like this would be a brittle integration that would be hard to test and would be hard to maintain in the long run. So instead, we've built a gRPC client on the internal CockroachDB APIs. So we're basically just calling the endpoints that the Cockroach CLI itself would be calling.

(14:21):

So what we do here is we call drain on a node without shutting down the node to remove it from node balancing. Then we wait for the standard period of time that we recommend all of our clients to use in their connection pools for the max lifetime of their connection. And this rotates all of the client connections away from this specific node. Then we initiate decommissioning and wait for the node to drain. Once that's done, we mark it decommissioned, delete the disc, and remove the node.

(14:49):

If we get in a situation where we are scaling from five parts to three parts, and we find out, oops, I mean we were looking at CPU utilization and thought we should scale down, but we actually don't have enough disc. We can reverse this and the operator will do what is in Kubernetes operator terms called, it's trying to be level based, which means that if you reverse the decision, it's going to start reconciling in the correct direction. So it'll recommission the node. If it's decommissioning, it'll cancel the drain and it'll bring the node back to life so it can start serving traffic again.

(15:22):

So for scaling discs, we took a little bit of a different approach. We still felt like even though we could automate the process that Frederik described, we didn't feel like that deleting the StatefulSets and leaving the parts orphaned was a good approach. So what we tried instead was, if we have the YAML document here on the right where we define the storage size for the cluster. If we change this to say we would like 64 gigabyte disks, instead, what we do is we spin up a completely new StatefulSet, the yellow one here, with larger discs, and then we simply drained the old StatefulSet, leaving only the new one with the bigger discs. So for the most part, this worked pretty well. The only problem was that this took quite a long time. So some of the things we learned while doing this was that particularly when scaling discs, this pattern of adding a lot of nodes at once put quite a lot of pressure on CockroachDB. It needed to plan out, how am I going to do rebalancing? This was a fairly heavyweight process, and when you immediately after joining all of these nodes started decommissioning nodes, this caused a lot of disruption internally in the clusters. Which meant that the decommissioning process had a tendency to get stuck, which meant that engineers still had to go in and manually move the specific range that got stuck or restart a node or help the process along.

(16:49):

So it also meant that previously scaling discs was a pretty fast operation. So you just deleted the StatefulSet, scaled the discs up, you reapplied the StatefulSet, and now this was all of a sudden a multi-day process. So we didn't always have to watch it because the operator was doing the work of scaling down the nodes. But it took a long time. The flip side was it was safe, and this also allowed us to actually decrease the disc size. Which we can set up a new StatefulSet with smaller disks and drain onto that. So CockroachDB helped us out there.

(17:19):

So the next thing I'm going to tell a little bit about is what does this look like in a multi-region setup? So our CockroachDB clusters are completely independent of the operator. This is, I mean, as it should be, we have separation between the control plane, which is the operator and the data plane, which is the cluster itself. And we've chosen to deploy the CockroachDB operator into all of the regions where we run Kubernetes clusters instead of having a centralized point. This means that the control plane itself is also resilient to a regional outage. If we lose Region X, we can continue to use the operator to do operations in Regions Y and Region set here.

(17:58):

The way we manage operations that might race something like...

Rasmus Bach Krabbe (18:00):

Which operations that might raise something like adding a user, deleting a user based on the custom resources that we pin this kind of operation to just a single region, which means that if region X is the region that is currently the manually assigned leader region, if it's down, it means we can create databases or create users. But usually if we have a regional outage, it's not the time when we're doing a lot of provisioning for folks. So it hasn't been a big issue. So the other issues that we've had to a larger extent is that when we get cross region traffic that breaks particularly in asymmetric ways, like Fredrick mentioned, this has caused quite a lot of pain. So if the links here start to degrade in either one direction or we start seeing packet loss in both directions, this has been a very hard thing to manage.

(18:51):

It didn't really matter how we set up the CockroachDB clusters. When connectivity internally in the cluster breaks, we see very large impact to the cluster. So this means that we've built out tooling to basically take out an entire region. So we have a CLI command that we can run that will take down every single CockroachDB node within an entire region, basically circumventing the issues that we get with network. This is the only real mitigation we've found to this kind of issue. So once we had this up and running, we have the operator up and running, we have it in multiple regions, we can orchestrate most of these operations, we were asked by the business to move a lot of our stuff into a new cloud provider. So this turned out to actually be a pretty smooth operation with the combination of CockroachDB and the operator that we'd built.

(19:40):

So the operator was a good place to abstract away cloud provider specifics. So we could write and test code and the operator to make sure that this works across different cloud vendors and give the storage engineers working with this just like a unified control plane to work with. So we don't care whether or not we're deploying on one cloud vendor or another. We deploy the same resources and everything else is handled internally in the operator. So what we chose to do here was that we set up some standard zone configuration via the operator, which tells CockroachDB to avoid placing data on nodes that are designated as gateway nodes. Then we picked a pair of regions at each of the cloud providers that were designed to have fairly low latencies, and then we set up a Cockroach cluster that had the gateway attribute to make sure we didn't put any data on there.

(20:33):

We set up the client applications and we started canarying traffic onto the new cloud provider. So this allowed us to verify that everything was working, CockroachDB was running well, the client applications were working on the new cloud provider, they could serve traffic and they could tolerate the increased latency that they were going to get when we were doing the actual migration. So once we had all of the traffic running on the destination cloud, we basically just did a rolling restart of the clusters on the destination cloud, removing the gateway attribute. This allows CockroachDB to balance the data across both clouds. Then we added on the cloud we're moving away from, pushing the data out and this allows us to clean up all of the stuff that we had running on the original cloud.

(21:18):

So this worked really well. We had a couple of clusters that needed special handling where we increased replication factors to support follower reads on both clouds at all times. But for the most part, this was a good process. We had an easy rollback by doing the canarying of traffic. So when applications weren't ready for this transition, we could canary the traffic flip back, do remediation, try it again without having any major impact or having any data that took a long time to move back and forth. So are we cloud native yet? It feels like we're kind of getting there. Cloud native isn't a nice and vague term, but at the very least we reduced toil by a lot by creating the Kubernetes operator. Management of users and databases were quite high toil operation previously. The team did a lot of this and it required a lot of manual steps, a lot of coordination with stakeholders on handing out certificates. It required storage team engineers to have key material on their laptops, and all of this went away and was automated away by the operator.

(22:22):

It also meant that we had reduced mental overhead on the team because everyone knew how the operator worked and that was kind of the only thing they need to know about. No one needs to know about the ins and outs of stateful sets unless they're developing something specific for the operator, and we can do this using unit and integration tests locally on a laptop and get everything up and running before we ship it into staging and production environments.

(22:45):

So this was really nice and it simplified our multi-cloud and our regional migrations by quite a lot. So what did we start doing next? Well, after having automated a lot of this stuff, we started looking into how can we be more efficient about how we run CockroachDB? CBU memory is expensive, disc is expensive, and we still had some remaining pain points on our disc scaling solution. It took a long time to scale discs and it had a tendency to get stuck. So we're wanting to try to improve on this story and Frederik's going to tell you a little bit about where we went from here.

Frederik Stenum Mogensen (23:23):

Thank you. Yes. So the third section running on Kubernetes, the next section operating, and the next session here, the last section is the right-sizing Cockroach on Kubernetes. Looking at efficiency basically, right? So we have a lot of different initiatives. The ones we have started or decided to take here, the first one is vertical autoscaling. So basically we started encouraging to autoscale all of our workloads vertically. So looking at how much CPU and memory the thing is actually using at runtime, and then autoscaling the requests in Kubernetes so we can ping pack our nodes harder, but still have enough headroom that we don't actually follow up and we have a spike coming in. So what we do is we look at the CPU usage over a long period. I think we're looking at seven days for the big period and a bit smaller period as well.

(24:19):

We look at the P50 usage and the P90 usage for the Cockroach nodes. We add in a headroom to ensure we can do all of the spiking that we need, and we also do it based on this. If you look at this graph here, you see all of the spiky lines at the bottom. That's actual Cockroach nodes and the actual CPU usage. And the green area on top is the recommended CPU request. We could maybe tighten this a bit more, but I mean, having the headroom to not follow up when we have a spike coming in is more money worth in my eyes than having a bit smaller cloud bill. If we look at this graph, we see an incident on the 6th of September where a workload changed drastically on the cluster. All the nodes spiking up using a lot more CPU. We see the autoscaler going, oh yes, you request more stuff for these one because these ones are running hard now.

(25:10):

After a couple of days, we decided to add in more nodes to the cluster and scale it out to have a bit more resources available in cluster adding in the yellow line here, which made all of the Cockroach nodes running less hard again and the autoscaler again recommending, okay, now we can reduce the request again and we can ping pack a bit harder because now we again see that the work is as we expect it to be. So this gives us the ability to auto-tune the request for our entire Cockroach suite and ping pack it much harder in Kubernetes, ensuring that we don't run a lot of hardware that we're not using, but we only run the hardware that we actually need.

(25:49):

The next project is the take three of this scaling, and this is actually a nice place to be. I don't see the big pings in this one. We can easily add a bit more on it, but this works very nicely. So for scaling up, we go back to the way we did it basically in the first one and depend on the cloud providers to scale the disc for us. But having the pain with the staple set being immutable, we actually take away the responsibility of creating the disc from the stateful set. So the stateful set will only create nodes now and we have a new operator, the PVC autoscaler, which is responsible for actually getting the PVCs and adding them to the nodes and startup. So if we have this set up here and there's a Cockroach node, it has a persistent volume, it has this of half a terabyte, if you want this to scale up, we define it on the cluster spec, this should be bigger.

(26:38):

The PVC Autoscaler will come in and look at the disc and go, okay, this is not matching the size state. We should increase the size of this one and it will call the cloud provider and go, I want to increase the size of this specific disk because it's supported by all the cloud providers we're running in. This just gives you a bigger disk. So this operation takes no time, a couple of minutes depending on the cloud, and this can be done completely without restarting Cockroach.

Frederik Stenum Mogensen (27:00):

And this can be done completely without restarting Cockroach. We had to restart all of the parts earlier, because they need to roll for the new [inaudible 00:27:07] to take ownership of them, but this actually works without a single restart. Now increase the disks of all of our Cockroach nodes without any disruptions. This is very nice, very cool in my eyes.

(27:20):

Going the other way is still a bit of a pain, because we can't tell the cloud provider to scale the disks down again. Apparently they don't want us to do that. So what we do is, we change the desire-state going, "I don't want a terabyte anymore. I'm not using it. I want half a terabyte."

[NEW_PARAGRAPH]The PVC-autoscaler will go in and they'll see, "Okay, this is a transformation I can't do on the cloud provider, but I can get you a new disk." So it'll provision a completely new disk of the right size. It'll deploy in a workload called copier, that will just take the bytes one by one from the old disk, and put it on the new disk. Depending on the size of the disk, this takes between seven minutes and four hours, I think. This is still much faster than the old, moving the stuff to a new staple set, which could take weeks for us. And when all of the bytes have been moved, it will go in, restart the part, and go, "You are now running on this disk. It has all the data you had before. It's just a bit smaller." So, this gives us a downtime for the Cockroach node because it can run while we do the copying in the current state of the operator. But this gives downtime for a max four hours for a single node, which is quite nice.

(28:35):

The last thing we are looking at in saving costs that we're bringing here is, running on Spotnodes. Spotnodes are cheap and fun. Spotnodes has a couple of gotchas that you need to consider when deciding to run on Spotnodes, such as the nodes may go away at any point in time. Not just one node, but a bunch of nodes, or all of your nodes may go away at any given point in time if the cloud provider decides that they need the capacity back to give to someone who actually wants to pay for it.

(29:05):

When this happens, depending on the cloud you're running on, depending on the version of Kubernetes, your pods may or may not get told that they're terminating for a long time. The pods just goes away and Cockroach will be confused that the node is gone, and have a bad time, and Kubernetes will be confused that a node is gone, and don't know if the thing comes back again. If they do actually get notified, the period they get notified in, is somewhere between 30 seconds and two minutes for having the shutdown down time. As Rasmus went through in the last section, we have a agreement with our stakeholders that they should roll their connections every five minutes, to ensure that we can do no maintenance without breaking connections. So we can actually terminate our nodes gracefully in this short window here. So this gives actual disturbance on the cloud, on the client workloads when this happens.

(29:57):

The last thing is that some clouds may actually require you to go in manually and delete the pods that were on a node that has gone away. Or at least wait for the timeout, which is 45 minutes, something like that, before Kubernetes decide that node is gone and isn't coming back, and will reschedule the thing for you. So for us, this works fine in staging and in testing.

(30:18):

It saves us a lot of money. It's a Free Chaos Monkey. It will force our stakeholders to also consider, "Well, what happens if we don't actually follow the agreement? The thing may go away. Can I handle this? Can I recycle my connections when I see it's broken?" And for production, we don't use it, mainly because losing all of our Cockroach nodes in a single-region is a very bad place to be in production, especially when you have single region clusters. We have a bunch of those as well. This gives very much an availability for everything, but it is a fun tool and it can save you a lot of money if you had the actual time to try it out.

(30:59):

So looking at the key takeaways as I see them, as we see them for this talk here, the main thing is yes, you can run Cockroach on Kubernetes, and it can run very nicely, and you can save money if you do it lean. Going a bit more into details, we can say that Kubernetes is a very powerful tool for automatization. It has the ability to do all the stuff that you need. It's a quite big AVI at this point in time, and you can do automations on top of it. Cockroach is a very flexible tool. It actually can tolerate all of the disruption that comes with running on Kubernetes. We use cluster autoscaler to ensure we don't have too many nodes in our clusters that are not doing anything. So we will bin-pack all the things out and the cluster-autoscaler will decide if some nodes are just doing a little bit of work, it should still go away. So this gives it a lot of disruptions to the workloads that's running in Kubernetes for bin-packing, and for scaling out again.

(31:55):

The operator pattern for the in-house operator we built has been a very nice help for this. It allows working very nicely with Cockroach and with Kubernetes, and it allows us to do the, "This is the state I want to go towards, just make this happen. If there's disruptions in Kubernetes, if there's disruptions from any other side, just ensure that you can go towards the state afterwards."

(32:18):

Again, Kubernetes works across multiple clouds, and has been a fine reflection on top of these. We use managed Kubernetes in all of our clouds, and this just works mostly out of the box. The managed disk, the network's disk, and the compute for the clouds also works very fine and compatible in the different clouds. If you can figure out which screw to hit. Yeah, as Rasmus mentioned, it's worth noting when you start doing high frequency optimizations as we do, that too many disruptions on Cockroach at once may leave you unexpected impact, right? So adding in 50 new nodes and draining 50 nodes at the same time will make Cockroach be a bit confused and have a bad day. And yeah, that happens. We also see when running multiple regions in multiple clouds that the network links between the regions is much less stable than the inter-region network links.

(33:13):

And just a few percentage of network failures, of packaged [inaudible 00:33:17], or something like that will lead to a lot of retries across regions. If you have a long loop between the regions, this will lead to timeouts in the SQL layer. So just a few percentage of failures in the network packages, or a few percentage of failures in the KB layer can mean that your SQL layer is virtually unusable. Especially if you have SQL that hits multiple regions at the same time. That's something worth looking at.

(33:40):

This becomes gray failures in that it works for much of the SQL write-off. SQL that's just hitting one region will be working perfectly. The SQL that has to go to another region, or move a lot of data will start failing. This is where we use some of the toolings that Rasmus mentioned, right? This region seems suspect, just kill the entire region off. It's fun side note, this actually makes Cockroach run better, when you kill an entire region off, because there's less traffic to do, there's less coordination, there's fewer nodes. I think just runs a bit better. So this is a nice tool to have in an incident in [inaudible 00:34:15]. I'm fairly certain if I kill the thing off, everything will run better, and I won't see any of my gray failures anymore.

Rasmus Bach Krabbe (34:21):

Yeah, sort of a side note on that, is that we've actually seen that when we take out an entire region, the CPU load doesn't just shift to the other two regions. I mean, because of the reduced load on Cockroach for quorum and for cross-region networking, we actually see that the increase in CPU usage in the remaining regions is actually pretty modest. Which means that the operation is fairly safe also from a capacity standpoint in our experience.

Frederik Stenum Mogensen (34:46):

Yes, and it's just fun getting up at the middle night and killing off a region and going, "Well, that fixed it." The last couple of things here to look at, is that when actually deciding to run multiple Kubernetes clusters and stretch your workload across those, the open source tooling for this is lacking. There's not a lot open source tooling that actually supports hitting multiple Kubernetes clusters, and orchestrating this for you. So at least for the things we have needed, we have needed to do a lot of bespoke tools. Lastly, running efficiently and looking at your cost, it requires a continuous effort when doing this in the public clouds. Yeah, well, thank you all very much for ...