EPISODE 568

[INTRODUCTION]

[0:00:00.3] JM: A message broker is an architectural component that sends messages between different nodes in a distributed system. Message brokers are useful, because the sender of a message does not always know who might want to receive that message. Message brokers can be used to implement the publish/subscribe pattern, and by centralizing the message workloads within the pub/sub system, it lets system operators scale the performance of the messaging infrastructure by simply scaling that pub/sub system.

Derek Collison has worked on messaging infrastructure for 25 years. He started at TIBCO and then spent time at Google and VMware. When he was at VMware, he architected the open source platform Cloud Foundry. While working on Cloud Foundry, Derek developed NATS, a messaging control plane. Since that time, Derek has started two companies; Apcera and Synadia Communications.

In our conversation, Derek and I discussed the history of message brokers, how NATS compares to Kafka and his ideas for how NATS could scale in the future to become something much more than a centralized message bus system.

[SPONSOR MESSAGE]

[0:01:24.6] JM: Software workflows are different at every company. Product development, design and engineering teams each see things differently. These different teams need to collaborate with each other, but they also need to be able to be creative and productive on their own terms.

Airtable allows software teams to design their own unique workflows. Airtable enables the creativity and engineering at companies like Tesla, Slack, Airbnb and Medium. Airtable is hiring creative engineers who believe in the importance of open-ended platforms that empower human creativity.

The mission of air table is to give everyone the power to create their own software workflows; from magazine editors building out their own content planning systems, to product managers building feature roadmaps, to managers managing livestock and inventory. Teams at companies like Conde Nast, Airbnb and WeWork can build their own custom database applications with the ease of using a spreadsheet.

If you haven't used Airtable before, try it out. If you have used it, you will understand why it is so popular. I'm sure you have a workflow that would be easier to manage if it were on air table. It's easy to get started with Airtable, but as you get more experience with it, you will see how flexible and powerful it is.

Check out jobs at Airtable by going to airtable.com/sedaily. Airtable is a uniquely challenging product to build, and they are looking for creative front-end and back-end engineers to design systems on first principles, like a real-time sync layer, collaborative undo model, formulas engine, visual revision history and more.

On the outside, you'll build user interfaces that are elegant and highly customizable that encourage exploration and that earn the trust of users through intuitive thoughtful interactions. Learn more about Airtable opportunities at airtable.com/sedaily. Thanks to air table for being a new sponsor of Software Engineering Daily and for building an innovative new product that enables all kinds of industries to be more creative.

[INTERVIEW]

[0:03:43.8] JM: Derek Collison, you're the creator of NATS. Welcome to Software Engineering Daily.

[0:03:47.8] DC: Thank you. Great to be here.

[0:03:49.1] JM: NATS is a message bus and we'll talk about what that is and why it was necessary to build NATS. Let's start with the topic of distributed systems and how a message bus fits into a distributed system. Why do distributed systems need a message bus?

[0:04:07.9] DC: Well, I think it's a great use case in terms of today's modern architectures of why is something like maybe HTTP, or maybe even GRPC, something that might not be enough. The old adage is what's new is actually old. Even in the late 80s when I started doing this type of work in early 90s, there's things like [inaudible 0:04:29.1] which said, “Oh, we just need to do request reply.”

Then came along message-oriented middleware, which was spearheaded by the invention of publish and subscribe, which in simplistic terms changes it from my conversation, a telephone conversation where I call you, ask you a question and hang up and call someone new, to a radio station where I just published out on a radio station and if you tune in to that, you can hear the radio broadcast.

It was especially important in financial industries where programmatic training was starting to take over – programmatic trading sorry, was starting to take over and whoever I made the telephone call to first started to have an advantage. When it's a human they didn't have an advantage per se, but the machine said, “Hey, I have information,” maybe a hundred milliseconds before the person that's last in line. That line kept getting exponentially larger as programmatic trading and access to stock quote data became popular within the early 90s.

Pub/sub took off and distributed systems started really taking off outside of those specific verticals when we really started embracing cloud and started looking at companies like Google doing horizontal scalability at a massive scale and how they would coordinate. I think Google for a while was in the mindset, and just as a heads up, I did work at Google for six years; 2003 to 2009 or so, where they would use either TCP/IP and protocol buffers underneath something called Stubby, at the time.

Now more in terms of HTTP, where we see cloud native and micro services starting to proliferate. I think what we're doing is actually going back to the future in the early 90s, where developers are starting to say, “Hey, we might need more than a simple request reply pattern. We might need more patterns to do event streaming and logging and different things where we're doing load balancing and teeing of services, where other things might be interested in watching those request reply flows that the system wasn't initially designed for.”

[0:06:30.4] JM: You've been involved in message buses for as long as almost anybody in the industry focused completely on message – well, I'm not focused completely on message buses, but in your career you have spent so much time on different message buses. When I think about the timeline of these high-volume message buses that get through into the conversation, there's TIBCO there's the older MQ systems, like RabbitMQ, ZeroMQ, we've got more recent open source solutions like NATS and Kafka, we've got cloud products, like Kinesis and google pub/sub. Why does this keep getting reinvented from scratch? Why is this so complicated? Can't we just solve the distributed message bus problem once and for all?

[0:07:19.3] DC: Well, I think that's a great question. I think I have the answer, but I've thought I've had the answer multiple times at TIBCO, with rendezvous which was more akin to the NATS design in terms of simplicity, but then at TIBCO I design EMS. EMS, I think was the first production system that combined pub/sub, queueing, transactions exactly-once delivery semantics.

I mean, truly exactly-once semantics, which for those in the know it's extremely hard to do and the penalty for that is usually not worth it. If you look at NATS, it's returned to simplicity and there was a couple driving factors. When an enterprise messaging system tries to do a lot of things, a lot of times a single client can make the system as a whole start to act erratically and to the detriment of everyone else.

If you look at extremely large-scale networks and how they try to be resilient, so you can look at the internet, you can look at the global cellular network, you can look at DNS, which is an interesting one, because it's been an attack surface for a long time in terms of very large-scale DDoS attacks.

A return to simplicity and very predictable behavior, even if the prediction and the predictable behavior is it will fail and chop you off is much better to deal with and rationalize a smaller scale, which then can scale up very largely, than situations where it becomes very complex, there's a lot of paths and control flows that go inside of the system trying to do different things.

For example, everyone says well, a messaging system needs have durable messages and transactions and at least once delivery, maybe even exactly-once delivery. If you think about it, the global cellular network if the call gets dropped, it doesn't do much about that, right? I'll call you back, or you'll call me back.

On the web, if I click on a web page and I get a 503 or something that doesn't look right, I usually just click reload or I hit do it do this again. I think there's beauty in being extremely simplistic only doing a certain set of things and still being able to build resilient systems, but that in the past needed a different skill set from systems designers that I think wasn't as proliferated as it is today. In other words, the system designers that I see today are extremely sophisticated. They understand how to do compensating transactions, idempotent messages for event stream state transitions.

All these other things which puts less and less requirements on an underlying communication protocol and puts it more back to the way the internet works, or again, a global cellular network works. For me, the transition over the last getting close to 30 years or so has been no, the messaging system has become more complex. The EMS system that was designed at TIBCO I'm extremely proud of and the people that worked on that with me and all is something that we look at as a great accomplishment.

There are systems that I then designed on top of that that are still running today and that people who are using these large transportation services, shipping delivery services, all other stuff are still, or the financial instruments, the stock market are still using today. They've been running for 20-some years.

That being said, even in those microcosms we were doing development of systems utilizing the messaging technologies, where everything that we had thrown in we actually convinced through code designing and co-architecting the solutions that again, have been around for 20-some years to not use certain things, because there was a massive trade-off that had to be made. What we were able to do is create a system where in certain failure scenarios, the ramifications were tolerable. In other words, we could say, “Okay, we get it. We understand.” Yet, we could scale from an initial design past five years with the initial design, which I think was something that all of these customers who I've still keep in contact with were extremely happy with the end result.

To give even one more example, Google, when I joined Google, a lot of things was purely distributed very lost-tolerant. Meaning, you could do a Google search and your searches might be slightly different, because if they didn't get the results back or whatever. That was okay, because the probability that I cared what's on page four of something just didn't get bundled into the response before it had to get out the door, which was at 200 milliseconds or less, that was all right.

When Google transitioned to Gmail, all sudden that became a very different problem set, where if I refreshed my inbox and all of a sudden, mail’s gone and I refresh and it comes back again, that's not good. Internally to Google, Google already had these problems around calculating ad revenue, right? When you clicked on something or if you had an impression, all those types of things. They were growing so fast that they actually couldn't do a lot of the things people feel that they should have done.

Without going into a lot of details, the earlier systems in Google were built such that they could keep up with the pace when everything was working. In certain failure scenarios, they would lose revenue. That was okay, because if they built the system to not lose any revenue, they couldn't actually collect the revenue that they were being generated at because it was coming in at such a fast rate. and I think in the in the world where we have so many resources available to us and in clouds and edge computing, which I think in the next 10 years will dwarf by two orders of magnitude, the resources that we have from a cloud perspective, there's a lot of trade-offs that need to be made that many people knee-jerk and say, “No, I need transactional semantics and exactly-once semantics and I can do 200 transactions a second, or 200 messages a second.”

The next thing you know, your service or software as a service takes off and it's trying to do a thousand messages a second and they can't. I think people are starting to become very savvy about that, they're starting to look at things differently that not all communication between all of these distributed system components should be request reply. They're looking for something that is simple, predictable and has multiple messaging patterns; again, request reply, pub/sub, distributed load balancing, eventing, event sourcing, those types of things.

I think that's why you're seeing an interest in NATS, an interest in Kafka I think is a disruption of the old enterprise models both in terms of older designs and cost, to be honest with you. I think it's a great time for people to revisit these two premises, that we need multiple messaging patterns in a world where it has more and more software complexity.

The software systems of today will never be simpler. They're going to become more complex, more moving parts. That HTTP is not the only thing that we need, and that simple actually scales, and you can build extremely resilient systems on very unreliable pieces underneath. Just look at the global networks that we have today.

[0:14:17.3] JM: There was a lot there. In terms of trade-offs between messaging systems, I think many people who are listening to this understand the value of a pub/sub messaging system and other features of a messaging system, whether we're talking about pub/sub, or just cues, just message queues, unidirectional message queues.

If we're doing the compare and contrast between different technologies, I think two pieces of open source technology that people are considering deeply are NATS and Kafka. If I understand correctly, one of the big trade-offs between the two is that Kafka has a lot more functionality, perhaps the persistence aspect of it, and NATS is deliberately simpler and does not have persistence. Would you say that's accurate differentiation between those two technologies?

[0:15:15.0] DC: Yeah, I think if you look at NATS, the core technology, right? It most wants delivery. Meaning, we fire-and-forget. We send the message. If you're around and you can receive it, great. If you're not, you don't. Again, if you're looking at the low-level technology like, " Wow, how can you build anything on that?" Believe it or not, you can build extremely resilient systems on that technology.

The performance of that technology, like on the iMac Pro that I'm talking to you on today, a single stream is about 20 million messages a second, it can multiplex a single server on this box, it can do 80 million messages a second with a single server. I mean, these iMac Pros are expensive, but relatively speaking that's great, great performance.

When you look at at least once, or exactly-once semantics, what that is essentially saying in layman's terms is that the messaging system will do its best to deliver to you. If you're not around, it'll hold on to it for you, and then when you come back online it can deliver to you. It will do some other tricks to make sure that it's to the best of its ability you only process the message once.

You could do a whole show on exactly-once semantics, but I mean, that's the general gist of it at the top. Right out of the gate, you might be able to do a very small number of messages to do that. One of the things that NATS does have is we do have something that was codenamed Stan on top of NATS. It is normal NATS clients operating as a high-level abstraction that essentially does log and log replay, similar to what Kafka does.

It has good performance on the clustering side, one of my big clustering things is if you have one thing and you add another thing, it should never be slower than the one thing, but that's an extremely hard, a serious systems problem to solve. I think we did a good job of it. In other words, you can send messages in, you know if they're there, any consumer can replay them from anywhere they want and at any rate and all that fun stuff.

I think we can do better. NATS streaming, or Stan was the code name for it, is more apples to apples with Kafka. The team is slowly starting to formulate a design that was started by someone we worked with that Apcera, Tyler Treat, who does a lot of work with NATS and the ecosystem and benchmarking called Jetstream. I think that will be where we want to go when there is a pattern that needs some – at least one semantics, or exactly-once.

Now that being said, going back, if you look at a developer, I do believe you're correct and that some developers are savvy to what pub/sub is and can utilize pub/sub for eventing, event sourcing, event-driven frameworks and architectures for sure. Even going back to a more simplistic design of a developer saying, “Hey, I need to make a request and get a response from these eight different services.”

This is a very common pattern for modern architectures, where you're doing one piece, but you're collecting information from lots of other pieces that are going to fill in the gaps of what you don't do. A developer can easily in my opinion say, “Well, I'm the developer and I know I just need to talk to you Jeff and get an answer back." I'm doing a request for apply pattern, I understand that.

Where someone who's been doing this for so long and I wish you is because of IQ, but it's more about scars on the back and saying that I always say, which is never make an assumption about what a message will be used for. It sounds silly, but I'll give you an example. The developer says, "Hey, I'm just sending a request to Jeff. I get an answer back that's all I need, so I can do HTTP, GRPC, whatever I want to do."

I would argue that once you move into production, ops is going to quickly say, "Well, how do we know if Jeff is responding quickly enough and how do we know when we need to scale Jeff up? If we want to scale Jeff up so that there's ten Jeffs that are forming like a McDonald's line so that we can do more requests per second and aggregate, how do we do that? Do we have to run a sidecar? Do we have to run a new load balancing element in our architecture that has to change from what the initial simple request reply is to production? What happens if we want to do anomaly detection, or governance model where we're doing regulatory stuff, where we have to just collect requests and responses and every once in an while check to make sure everything's okay?"

When you look at NATS, NATS underneath the covers is a pure pub/sub system. Meaning that a request reply pattern is extremely fast, but it has no indication of location at all; doesn't matter where the Jeff is. It can automatically load balance transparently without me or the original Jeff ever going up and down. We just add more Jeffs and it just works. We can add anomaly detection by just saying I have another subscriber that's watching these requests. It's not going to answer Derek, but it's looking at something about the requests, looking for anomalies.

Then on the other side, there's another responder or another listener on the request that's just looking for regulatory breaches, something like that. You will see, especially the load balancing one where it's like how do we scale the Jeff service up and does that affect the clients Derek? NATS allows you to do that from day one. You stay in your request reply pattern, but you go into production and the binaries never change and you don't even change that I'm already running and you're already running.

Ops has the ability to control load balancing and regulatory stuff and anomaly detection, all different types of things. I think we're starting to see obviously dev and DevOps have become extremely popular. I think you're starting to see dev who are the DevOps chop saying, “Yeah, it's a request reply pattern.” I already know, because I've done this in the past when I deployed the service to production, more people are going to be interested in that request and the response than just me and Jeff, to start out with.

[SPONSOR MESSAGE]

[0:21:13.8] JM: We are running an experiment to find out if Software Engineering Daily listeners are above average engineers. At triplebyte.com/sedaily you can take a quiz to help us gather data. I took the quiz and it covered a wide range of topics, general programming ability, a little security, a little system design. It was a nice short test to measure how my practical engineering skills have changed since I started this podcast.

I will admit that, though I’ve gotten better at talking about software engineering, I have definitely gotten worse at actually writing code and doing software engineering myself. If you want to take that quiz yourself, you can help us gather data and take that quiz at triplybyte.com/sedaily.

We have been running this experiment for a few weeks and I’m happy to report that Software Engineering Daily listeners are absolutely crushing it so far. Triplebyte has told me that everyone who has taken the test on average is three times more likely to be in their top bracket of quiz scores.

If you’re looking for a job, Triplebyte is a great place to start your search, it fast-tracks you at hundreds of top tech companies. Triplebyte takes engineers seriously and does not waste their time, which is what I try to do with Software Engineering Daily myself. I recommend checking out at triplebyte.com/sedaily. That’s T-R-I-P-L-E-B-Y-T-E.com/sedaily. Triplebyte, byte as in 8-bytes.

Thanks to Triplebyte for being a sponsor of Software Engineering Daily. We appreciate it.

[INTERVIEW CONTINUED]

[0:23:11.5] JM: I think I want to bring things to a concrete example. Derek, let's say that we're starting a cryptocurrency trading company, because trading is the domain in which you saw messaging used the most in your early days at TIBCO, and I'm sure you've seen messaging deployed in latency sensitive environments since then with the lineage of different messaging systems.

I think, the basic ideas that the trading companies are trying to tackle are fairly similar to what we had even in your earliest days at TIBCO. If we're starting a cryptocurrency trading company, we're going to get a bunch of feeds from different exchanges, and we have a bunch of traders who are each operating their own nodes where they're doing their cryptocurrency trading. How would a NATS deployment work in this context? Why would we want this message bus and how would we deploy it?

[0:24:13.7] DC: Sure. This goes back to 1988, right? The first thing we would want to think through is what is the user experience of the crypto-trader? What do they care about the most? I'll try to make it simple and we can debate, but I think for the most part they care that they have the most up-to-date information. All right, so they want to know the price of Ethereum right now, or the price of Litecoin, or Ripple, or Stellar, or whatever those are, Bitcoin Cash, Zen Cash.

Right away if I am the one that's responsible for delivering those feeds and I am making individual connections to every single client to do that, the person who's at the end of the line right is now getting information slightly behind the person that I talked to first. You want to start looking at a pub/sub model there where I publish it, the infrastructure is very efficient even if the last model is TCP/IP connections and you can do some embargoing; you're trying to be fair to all clients, irregardless of what they do.

We get it out the door about the exact same time, so that if you are running a little program that says, “Oh, the price dropped. I want to buy some,” it's considered fair. Now here's where it becomes very interesting, because again if someone says, “Oh, I might need a messaging system.” A lot of times people immediately say, “Well, I need Kafka or I need something that does durability.”

What I would argue is is that again, going back to that user experience, if you as a user let's say start to slow down, is it important that you see every single update of the price, or that the price that you have in your hand is the most relevant to the price it is exactly now? We saw this a lot in the late 80s, early 90s, all through the 2000s, where a publisher might be publishing fashion as the subscriber can keep up.

If you're publishing something that you consider to be idempotent, meaning you just need the last value, you don't need everything in between, you actually want to drop some of those things on the floor, because you don't want to be saying, “Oh, the price is this,” but it's actually not this, because you're slow and you're two seconds behind and you're saying, “Wow, the price just dropped,” but actually when you go to execute the trade, it's actually already risen back up, but you're looking – essentially it's like us looking at the stars, right? It's actually some time in the past and we don't know.

When you ask a messaging system do more than just fire and forget, it actually is working against you in that user experience scenario. You don't want it to try to deliver every single message, you want it to get you the last message that it has the best way it can. Again, it's not about as much the messaging system or the systems on top, what we try to do at Synadia who's the company that's spun out of Apcera is what is the user experience that you're trying to obtain and under what situations are we doing the right thing to maximize that user experience, or are we doing the wrong thing? A lot of times, enterprise messaging systems because they try to do so much actually work against the user experience that you want.

[0:27:11.6] JM: Why does persistence trade off against the – I guess, I'm trying to better understand what causes these tradeoffs? Because I think of persistence as a disjoint feature from the at most once fire-and-forget at processing. I can imagine a separate thread that's doing the persistence and from the forwarding of a message, like a feed, a new message from a feed comes in to your NATS instance and you could, okay obviously you want to fire and forget it, so you've got these feeds that are coming into your NATS instance and you want to send a message to all of the subscribers to the topics within that feed that are relevant to those subscribers, or maybe they want the entire feed, let's just assume they want the entire feed. I'm having trouble understanding why it would be a penalty to also persist those messages asynchronously?

[0:28:17.9] DC: Well I mean, a couple things, right? Fire forget system, like NATS one of the things that it does extremely well is that it's very predictable even as resource usage. Most NATS servers run for years at a time and they consume 20 mega memory and that's it. We don't do a lot of buffering.

When you're starting to store stuff on disk, it's like, well how much disk can I use? Is it a disk? Is it a service? There is time waiting, and now of course you can make it asynchronous, you could put it into a co-thread, Go routine thread, whatever you want to call it for sure. Let's pop up a level, so looking at what are the use cases where we would want any of that type of functionality, and they do exist. It's not that they don't exist. I just think in modern systems, the probability of those exists is lower than most people think.

When fire and forget is okay is if you and I are on line at the same time, my little words I just created the message, you're around and so you'll get it, or you won't if you're too slow or something like that. Essentially, it's the semantic of we're processing connected and online and healthy at the same time, you get it. When I say I want to save the message, the assumption is that the reason I'm doing that is because you're not online right now, or your rate can't match mine, meaning I have to do some buffering and you want to utilize multiple mediums to do that, so you don't have all your pressure on memory, which is still probably the most expensive compute resource, believe it or not, is memory, it’s not network anymore.

Or I want to process totally in a batch at my own rate sometime in the future. I don't have to be present when the message is being sent. That's really the only reasons that these other subsystems and messaging that can move from at least once to – or sorry, at most once to at least once or exactly-one even exist.

They do exist. In other words, there are reasons where you say, “No, I want you to just record every request. I can't get to it now, but once a month were going on a batch thing that might be a little bit slow, takes two days to go through, but the system can redeliver all those messages at the rate that the batch program wants to do it and they do it once a month and then they all say throw all the messages away.”

Those are legitimate. I mean, I understand those. Where I push back a little bit on communities is that, it's like the new shiny thing, which I understand. It's like, "Oh, new shiny. I want that." A lot of times that's not necessarily the right thing. Funny story, really quick is I remember the Z24 came out. Chevrolet Cavalier Z24, it had at all digital dash, this was 1987 or something like that, and I had to have one. I was so adamant I had to have one.

My father finally acquiesced and got me one, and they had a defect where the mounts on the dashboard would vibrate free and the dashboard would just turn off. You had to pound on the dash and all this other stuff. It was interesting, because I wanted the dash because I thought it was really cool, but when it went out the ferry was totaled, I had no clue how much gas I had, how fast I was going, anything. I think it's all about trade-offs.

Items like Kafka, or persistence and replay and all those things exist? Absolutely. Do they exist in 90% of the use cases of how online systems are trying to coordinate and communicate and deliver a software-as-a-service? No, I don't believe so. If you look at even Google, when you do a Google search, there's no such thing as a messaging technology that's storing messages and trying to replay them. They don't even have time to store a message, right? They've got to get in and out the door in under 200 milliseconds. At least that was the rule when I was there.

What they do is actually say, "Screw this. I'm going to ask 15 Jeffs for the answer, and the first one to come back the fastest wins, because that maximizes the user experience." If the fourth Jeff just doesn't even respond to me, I don't care. It's whoever responds fastest. I actually optimize NATS. One of the patterns that NATS does underneath the cover is that most people don't know about or don't care about is I can optimize around asking a very large set of N a question, when I only want one answer. I want the fastest and best answer to come back to me, but I don't want my client library spinning trying to throw 10,000 messages away every single request. NATS actually is optimized to do that extremely efficiently at the client layer.

[0:32:32.6] JM: Okay, so you're talking about a good use case here from Google, that I've heard you talked about before, which is clients, or user submits a search query, and the search query hits Google's infrastructure and gets fanned out to different portions of the Internet, the search query gets processed across these different partitioned portions of the internet that have some union with Google's indexing of the internet.

In each of these partition sets of the internet, who knows if this is how Google search still works, but this was your anecdote from a little bit of your time spent there, but each of those partitions, the fanned out partitions, you get you get a response, or you get a response from maybe you need just three responses in order to aggregate something asymptoting towards your perfect search query response. Then those things get merged, and then handed back to the user.

Is this something where Google needs a message queue, a pub/sub message queue? Do they need a piece of middleware to get that query and to fan that query out in a pub/sub fashion?

[0:33:44.1] DC: I think they could take advantage of it. Again, both of our information is I'm sure dated, but just for the listeners Google used to have the notion of okay, you're doing a Google search. I'm going to search for Elon Musk. Google shards up the whole entire index of the web. There's pieces that only deal with a certain number of web pages and there's tens of thousands, millions, hundreds of millions of maybe today. Then what they do is for each shard, they run replicas, and the replicas is the redundancy where I can ask all of the replicas for the answer and the fastest one back I take and I chop the other ones off and I don't care. That again maximizes that user experience.

Now what's interesting is is that Google again, dated architecture, so it's probably very different now, it didn't ever try to replay requests, or anything like that. It made them redundant and they sharded them and then they logged certain information, and logs was where you saw the batch processing about getting better at search, or looking for things that a lot of times people with a Kafka system are trying to do both at the same time.

Google could never actually process those logs, put them together and then run something to look at them fast enough to support the number of people that use Google as a service and want the results fast. I don't know if we still do it today, but I remember a time in mid-2000s, late 2000s, where if you wanted to see if the internet was up, you just typed in google.com, and people just would search for anything. Meaning, they felt that Google was about as resilient as you could get, yet underneath the covers they didn't use any of that.

Now, the reason I say that they might benefit from pub/sub is if they use something like NATS, where it's fire-and-forget they would have the speed and the latency characteristics of collecting responses and getting the page back to the user, but they could also have other processes running that are just snooping on this, watching it without having to wait for the log files to be written, collected and then processed there.

If they wanted to do that more real-time and in-line, they would have to, just like people who use request replies the dominant paradigm and pick their architecture based on that, would have to add additional complexity to their load balance, or sidecar, or tee-off request streams, so that they could do this. Whereas, something like NATS just has it built in. It's built by design to allow that from day one.

[0:36:12.9] JM: When you have something like NATS – we're talking about two circumstances. One is this cryptocurrency trading company, where you've got feeds coming in and you've got traders that are subscribed to those topics, the cryptocurrency feed, exchange feed topics. If there's a network partition and there's some period of time where the feeds are maybe the feeds aren't getting updated and sending messages to NATS, or maybe there's a partition between NATS and the traders so that messages are getting dropped between the traders and the NATS cluster.

Then once that network partition goes away, you are reconnected and you'll get non-stale data at least, I would assume, because NATS has fired and forgotten and it fired against a network partition and failed, but in the other circumstance where we're talking about a search query, that's a situation where we probably would want to retry or we would want to at least give the client some knowledge that there has been a network partition, sorry your message was dropped at some point along the way. This is this seems to me like one place where persistence within the messaging system could be useful.

Maybe you could just give an explanation for how network partitions get handled in a messaging system. Do you need smarter clients to deal with that? Do you need clients to potentially do retries? What do you need to do there?

[0:37:53.8] DC: Yeah. I mean, network partitions are something that all distributed systems have to account for. Believe it or not, when things fail and you know that they failed, that's actually not trivial, but it's not super hard to design for. It's when things actually slow down. They're still there, they just become very slow and that slowness cascades through a system. Hence, the Adrian Cockroft came up with the notion of latency monkey, which I said was way better than chaos monkey, because that was – that's a 101 class. You're at a 600 level when you're doing latency stuff.

In your case the network partitioning, the one thing to think about, again looking at the user experience and the data that's being sent that I at least try to push clients and customers that we interact with on an architectural level say, "Well, what is the data representing and what type of data is it?" For distributed systems, I really want this notion of a piece of data as idempotent. Meaning, that you don't have to receive anything before, what you get in your hand gives you the complete state of the world.

Now in the early 90s, the networks weren't fast enough to do that, even for stock trades, our first stock prices. We had to do initial value caching, so when a new subscriber came up, we handed him a big blob of the initial value, then there was a delta stream on that. When you look at streams and you say you have to receive all 10 messages to have the same state as I do versus if I only receive message 10 and you receive message 1 through 10, you and I both have the same view of the world, that changes the requirements underneath and it makes the serial architectures a lot better. Because you can just keep repeating message 10 if you want, it doesn't matter. Meaning, if you never receive it, I just keep sending it out every second if I don't have anything new to send.

Now on your architectures where you don't have to store messages, you don't have to retry them, it becomes a lot simpler in my opinion. For Google, again I can't speak directly for them but to the best of my knowledge, they would never try to store a request and do retries per se. They would record the fact that they might not have complete results for a query and that that's not good, but they've built so much redundancy and sharding into there that – and I would imagine those redundancies right across different network paths, so that if you have a network chop, well I have a shard, but there's a hundred replicas in there and 90 of them are still up. Meaning, I'm going to get the answer back from one of those 90 pretty quickly.

I don't believe that they do that. They will possibly say, “Hey, I'm having a serious problem. I can't see anybody. I'm just going to shut down and not even take any user requests,” which is in my opinion the right thing, fail fast, right? Don't try to hold on and try to retry and try – just fail fast and everything upstream that Google has partitioned, meaning you type in a Google search and you get something weird, back what do you do? Command-R, right? Try it again. What the heck's going on? The next thing you know it works fine, but it's rerouted behind the scenes.

Very large-scale, very resilient systems I think are predominantly built on things that aren't as resilient underneath the covers. Now that being said, you can't do credit card processing on that per se. Even Cloud Foundry, Cloud Foundry was a system I architected. It's very different in architecture now. The pivotal team has done an amazing job at driving it to the level that it is, but even the early design had no message replay, no durable message. It was built on the original version of mass, which was fire-and-forget.

What it did was it say, “Hey, I'm going to record that Jeff wants to run this app and then I'm going to send out a request for someone to run this for me. I don't care if they respond. I don't care of anything, I just send it out.” The system itself has another component that all its job in the world is to do is to say, “Here's the intended state and here's what I'm observing, because it's watching the messages fly back and forth.” It's going, “Jeff should be running, because I see that in the intended state, but I actually don't see Jeff running in the real world.”

Instead of it doing anything on its own, it would just simply complain back to me as the original person saying, “Hey, I want Jeff to run,” saying, “Hey, by the way, Jeff's not running.” It would tap me on the shoulder and I would reissue the request, but exactly the same way. I never tried to replay it, I didn't want anything stored, and yet the system was extremely resilient even in the face of multiple failures.

Again, I think the audience of this podcast and the general audience is becoming so much more aware of patterns in the application level that more needs something that's always on, it's like a dial tone and it does what it says it does in a predictable manner, versus the end-all be-all with every feature back then.

[SPONSOR MESSAGE]

[0:42:50.8] JM: Users have come to expect real-time. They crave alerts that their payment is received. They crave little cars zooming around on the map. They crave locking their doors at home when they're not at home. There is no need to reinvent the wheel when it comes to making your app real-time.

PubNub makes it simple, enabling you to build immersive and interactive experiences on the web, on mobile phones, embedded into hardware and any other device connected to the internet. With powerful APIs and a robust global infrastructure, you can stream geo-location data. You can send chat messages, you can turn on sprinklers, or you can rock your baby's crib when they start crying. PubNub literally powers IoT cribs. 70 SDKs for web, mobile, IoT and more means that you can start streaming data in real-time without a ton of compatibility headaches. No need to build your own SDKs from scratch.

Lastly, PubNub includes a ton of other real-time features beyond real-time messaging, like presence for online or offline detection and access manager to thwart trolls and hackers. Go to pubnub.com/sedaily to get started. They offer a generous Sandbox to you that's free forever until your app takes off, that is. Pubnub.com/sedaily, that's P-U-B-N-U-B.com/sedaily.

Thank you PubNub for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:44:34.2] JM: Let's talk more about the time at Cloud Foundry when you were at VMware. You spent a lot of time architecting Cloud Foundry, What does a system like cloud – for the listener, Cloud Foundry is a system for our orchestrating your infrastructure, orchestrating VMs, getting distributed applications up and running, getting different distributed applications communicating with each other. We've done shows on Cloud Foundry, but what does Cloud Foundry need from a messaging system?

[0:45:06.9] DC: I can only speak to the original design, which is we relied heavily on NATS to do command and control, allow for location transparency. We had no clue –

[0:45:16.7] JM: What's command and control?

[0:45:18.5] DC: Command is I'm sending a command saying, "Please run Jeff, or I want you to change your state to X." Command and control, so in other words it was sending requests around the system saying, "Hey, I need these things done. It wouldn't necessarily wait for an answer, or try to do replay if it didn't happen." The location transparency in cloud is –

[0:45:37.6] JM: Sorry, just to ask a naive question. For those who are really just catching up to the idea of messaging, maybe they haven't worked with it directly, why would you use a piece of middleware infrastructure like NATS, rather than just sending an HTTP message to whatever piece of infrastructure you're trying to send a message to?

[0:45:57.6] DC: Yeah, I mean, that's always a great question. In the late 80s, I mean, early 90s, especially early 90s, I could walk in almost any financial house in Wall Street and go into their server room and point to the exact rack, exact machine that TIBCO stuff was running on and I could tell you what the IP of that thing was, meaning it was a known entity.

As we've transitioned to cloud, we have two things that are going on. IP's change quite a bit and we obviously want to be able to horizontally scale different tiers of an architecture. Meaning, we want to have more than one. If we wanted to do Cloud Foundry in that respect, it's doable, it's not impossible, but now every single front-end would have to poll to see where all the DEA's are, have to figure out a way to keep that list up-to-date, have to figure out a way to figure out if we want to do load balancing, or if we actually want to do pub/sub. Meaning send it to all of them are only send it to one randomly, or some load balance algorithm, like Round-Robin. All that accounting is going on.

With something like NATS, you don't have to do any of that accounting, all you have to know is I send requests to this subject. That's it. I don't care where they are, I don't care if they're zero, one, a million, I don't care if they're going down, going up, moving around, I just know I send one thing. The application logic that is saying, "Hey, I got a request to run Jeff is very, very simple." It's very clean and flows very well.

Also, you can utilize different patterns, right? Pub/sub, request reply, load-balancing, all with the same technology. Whereas if you're doing HTTP and you're saying, Hey, I want to do load balancing, well then there's something in between you and that endpoint that is doing that for you, right? ELB, or nginx, or Google load balance or any of those types of things. It just makes the architecture I think cleaner and simpler.

Again, those aren't siloed paths of communication. Again, looking at Google you could argue again dated information, but that the communication wasn't siloed because we logged it, and so then after the fact we could say three days later when we collected the log and analyzed them, “Oh, Derek sent the request to Jeff to run something for them.” The world is becoming more real-time, and even Google I think probably doesn't do this anymore. They're able to react in real-time to anomalies and requests and response and latency and all different types of things.

Pub/sub just naturally fits into that, because if I say, “Hey, I'm sending a request to Jeff and I want him to run something, I can easily run another service that's all its job in life is to watch the latency of how long it takes Jeff to respond.” If it's too long, I create more Jeff's. If it's short and I'm over-subscribed, I can create less Jeff's, pair you down. That naturally flows into an architecture where the original developer said, “I'm just asking Jeff to do something for me. It's just simple request reply,” but then ops and DevOps can actually do more sophisticated things as they see fit with very advanced type of sophistication.

Again, it goes back to that early premise that I wish it was mine, but it was taught to me by some of the original inventors of publish/subscribe technology was never make an assumption about what a message is going to be used for, because by tomorrow you could be wrong. Meaning, oh, ops doesn't care necessarily about Jeff only receiving it. We need the latency performance monitor to be able receive it and we need the anomaly detection system receive it. We need the regulatory confirm the stuff. Pure pub/sub systems even that had those very low latency request response patterns on top, naturally flow into those without any disruption. It's just easy to actually have those on.

[0:49:40.8] JM: One way to calcify what you're saying there, we wound up where we are, we were talking about command and control within Cloud Foundry, so maybe one piece of infrastructure wants to send a message to another piece of infrastructure that says spin up a server, and you could imagine, oh okay, the only thing that needs to know about spinning up a server is whatever piece of infrastructure is in charge of spinning up servers.

That may not necessarily be true tomorrow. Maybe tomorrow, you have a – somebody stands up a logging system and they say, “Hey, I also want access to all the messages of people who are saying I want to set up a server. How do I get all those messages?” If you don't have a middleware piece of infrastructure that is all those messages are being centralized in, or they're all passing through, then it's not going to be – it's not going to be simple to get access to all of those messages. I think that that maybe drives home the point of what you're trying to articulate.

[0:50:41.2] DC: Yeah. Think about it, original inbound request which is formula the socket, you could tee that. In other words, so let's say we wanted to solve that problem, now again, dated information, but the way Google solved that problem was is that they would log, write everything that the system was doing and those would slowly be asynchronously collected and then different jobs, sawmill jobs at least at the time, would run over those logs looking for things.

That was at some point in the future. Could be maybe 10 minutes, could be 2 days could be whatever. In the case where you're saying, hey someone else might need to see this request. I can actually – if their request is HTTP, if we have a load balancer already in place in production, we could then tell the load balancer to tee the request, right? Send it also to someone else, so they can see it. They can do it through a log, through some other mechanism.

The response is private. We can't see that response necessarily, so we either then have to say we have to snoop low level TCP/IP traffic, or we have to change the responders client code to say, send it back through the HTTP socket that we got the request on as a response and also send it over here or write it as a log.

Again, every time you are changing components to add functionality at scale, that becomes in my opinion problematic. If everything's pub/sub, you just get all that for free. There is literally no changes, you only have to take running code down to add additional components to start adding functionality to it.

[0:52:06.4] JM: Let's talk a little bit more about NATS specifically. You wrote NATS in Go instead of all the other language choices you could have made. Why did you write NATS in Go and what are the implications there for that language choice?

[0:52:24.1] DC: Yeah, so the original NATS was written in a weekend in Ruby actually. Cloud Foundry was originally written in Ruby. I still have a place in my heart for Ruby as a development language to tinker around with ideas and put things together. It didn't serve our purposes as much in production. The dependency management stuff was challenging. For the most part, Ruby was not any type of performance hindrance, but sometimes you get things that you try to want to work around.

We were also using a vent machine, which was a C++ asynchronous event loop inside to do some of the multiplexing on the I/O and stuff. Go came out the time, I had dealt with Java dating myself when it was called Oak. I knew about Java, I knew about the expert levels of tuning the garbage collector and everything's on the heap. I remember spending two weeks of my life at Google that I'll never get back trying to tune one of our applications just around the heap, because it was – it had issues, let's say.

Go it just come out, it was brand new, its compiled into a static binary, so no dependency management. I was like, “Oh, I like that.” I still love C. I'm just too old to program it anymore, so it felt like C, but it had memory management, but the kicker was that it had real stacks. If something was on the stack, it wasn't on the heap and it wasn't applicable to the garbage collection.

When Go first got released, it actually had a very simplistic garbage collector, a very simple mark-and-sweep. It worked very well, because the programmer, unlike Java, was able to control your own destiny. I could put things on the stack, versus only having them on the heap. I liked all of that, the first app I ever wrote in Go 0.52 I think it was, was the test that stacks were real and that I could actually do that.

Then I decided as we moved from NATS as an individual project that I ran starting Apcera and we were going to utilize NATS heavily as well, similar to Cloud Foundry to do the command and control plane and location, addressing and discovery, that I want to rewrite it in Go. We made a big bet on Go. I felt that it had a good chance of being the de facto implementation language for cloud infrastructure, systems infrastructure type stuff.

You can debate whether or not that came true, but I do believe that the Go ecosystem has accelerated and gained in popularity, and the mechanism and sophistication of Go the language and the community was a great bet for us to make. It's worked out very well. The original Ruby NATS server, again that was written in two days, so it wasn't complex, but the same protocol was designed in that weekend still runs today. It could do a 150,000 messages a second. Now today, we're looking at the iMac Pro doing 80 million messages a second.

Go has a lot to do with that. It helps quite a bit. It allows you to take advantage of multiple threads of execution, multiple cores, right? The new scaling model is horizontal, not vertical as we all know. We really got a lot of bang for our buck there. Now that being said, most every client language known to mankind has a NATS client for it. Just because a server is running in Go, it doesn't mean you have to pick Go.

I think we have really popular, at least in terms of download statistics for our node client, our c-sharp client, Java client, we do have a Go client obviously. Python and Ruby are very popular as a C, right? As we get closer and closer to the edge, people care about how much memory and resources the endpoints are using, because you got to power that. If it's not connected to a power source and it's a battery, writing something in Java versus taking some time and writing it in something like C could mean battery can last another year longer, or whatever.

Yeah, it was a good call for us. I still am a big fan of Go and I think the ecosystem has proven that out. Is it a perfect language? No, of course not, but I can tell you that one thing that surprised me that I didn't understand when we made the early selection is when you go back and look at it now, I can go back to code I wrote a year ago, two years ago and I pretty much understand what I was trying to do.

When I would go back and look at C code bases, or Ruby code bases, especially because of the meta-meta stuff that you can do, I would spend hours just trying to figure out what I was trying to do. I couldn't remember and the code wasn't helping me dissect what I was trying to do, even with comments that we're in there. I like that about Go, its ecosystem is great, for performance measurement and benchmarks and all kinds of stuff. Good lucky choice, but a good choice that we made.

[0:57:01.0] JM: Indeed. NATS joined the Cloud Native Computing Foundation somewhat recently, so the Cloud Native Computing Foundation is the organization that houses Kubernetes among other projects. How does NATS fit into that ecosystem, and how are you seeing people use NATS in Kubernetes-based deployments?

[0:57:23.0] DC: The decision to join the CNCF was one that we've been contemplating for quite amount of time. I was very fortunate, I was one of the founding board members of the CNCF when it originally got started. Was privy to what the early charter was, how it changed, how the TOC came about and project selection criteria and governance models, which I'm very thankful for. The reason for the CNCF was we really feel that there is a potential for a massive opportunity, not as much in NATS as a tech, or pub/sub as a tech.

If you look at the precedents that have been set where we moved from multiple ways to do communications on networks to TCP/IP, which wasn't perfect but the result of us standardizing on one was immense. The World Wide Web where everything was connected and we could share things and stuff was immense, and it's still growing. The world of connected IoT edge devices, last mile edge computing is going to amplify that even more.

Yet, we don't have a consistent way for things to communicate. Take that along with I think having some attention brought to NATS, and I think a good way, I think it's great technology, it's very reliable. I mean, it runs nuclear power plants that they've been running for years and don't pay support, don't do anything. It was just a natural fit to go in. With micro services, cloud native moving into more complex micro services and us starting to see people wanting to look at and entertain multiple messaging patterns between services, not just HTTP and request reply, it felt a natural fit.

The CNCF is great and they've done an amazing job at promoting and we're going to have a big presence at KubeCon in Copenhagen and also at the following QCon in December. Rolling back, I think the CNCF does a great job at drawing attention to technologies that can be part of a toolbox, a toolset for modern system architecture experts to utilize to put together these disturbing systems, which for the most part every system designed today is a distributed system. There is no one big fat binary that's running everything anymore that we used to have.

Now that being said, we also looked very heavily, or at least I've been looking very heavily at the challenges that open source software has in commercialization. I think they are real, and I think things are – there's opportunities to do things differently that move open source projects away from traditional revenue streams, which I don't believe will continue at least in terms of growing new companies, like support and education and training and maybe professional services. We have to figure out a way to not make it a charity project.

All right, most popular open source projects are a charity project. They're asking people to pay for development, if it's not part of a larger organization. One of the things that we've looked at heavily is how do we take something that is like NATS, is open source, very liberal license, it was MIT, now it's Apache too with the CNCF, it's fairly easy to get up and running and it's very resilient and very predictable and it just runs for years. How do we actually commercialize that? How do we actually make money off that when it's so easy for people to just run it? Half the people don't even monitor it, because if it goes down, it'll just pop back up and all the clients reconnect, the cluster topology reheals itself.

Where is their value in a commercial effort around there? We don't have any of the perfect answers, but Synadia is trying a very different approach. We believe there's an opportunity for a global utility that has security by default, isolation by default, but is a global infrastructure, like the global cellular network, right? We don’t own our own cell towers, we just utilize whatever cell tower is closest to us and the system just works, but for the most, part people aren't eavesdropping on our conversation. We know that's not true, but you get the point I'm trying to make.

We think there's an opportunity to put a global communication system out based on NATS core nets, which is very simple, like the internet fire-and-forget, that's secure by default. We can't wait 15 years for the green lock like we did with the internet. It has to be secure, it has to understand not only authentication, but authorization. Meaning, not only that I know it's Jeff, but what can Jeff do?

[1:02:00.4] JM: This is like Google cloud, pub/sub, or Kinesis, but even wider in scope?

[1:02:07.9] DC: Yes. Cross-cloud, cross-edge, cross-edge devices. Single URL to works anywhere in the world. What's interesting is I want people to treat it like utility. It's not a charity project, it's not free, you have to pay for it, but it'll be cheaper than you trying to run even a very, very small installation by yourself.

Now, what we want to do after that is we want to incentivize very specific behaviors to make the network valuable, which is we want to incentivize it to be decentralized by design. Have other people run servers, not only us. We want to incentivize publishers to share data that people find useful and we want to incentivize the OSS community to continue to develop interesting technologies that can value, can improve that network.

[1:02:51.4] JM: Okay. I definitely did not expect the conversation to go in this direction, but this is beginning to sound like some of the ICO companies I've interviewed recently.

[1:03:00.1] DC: Well, the ICO is interesting. It's a tainted word right now.

[1:03:04.1] JM: It is. I mean, it sounds like a project that is ripe for crowdfunding.

[1:03:11.1] DC: Possibly, but again, because ICOs are tainted and there's some regulatory concerns, the biggest thing is that the high-level plan for us to execute on is consistent, whether the method of payment is a traditional fiat, or a token, which is if you use the system, it's like your cell bill, or your water bill, or electricity bill, or like trash bags, you pay for it. It's very little, but you pay for it, it's a utility. From the consumption standpoint, you pay and we collect all of that money, again, not us, but it just a totally transparent, decentralized, possibly on a blockchain, we collect all of those fees and then we merely turn around and redistribute 80% to all the operators, 10% to the popular publishers and 10% to the OSS community.

Now there are secondary tiers in that. The 80% blob that gets stuck into a ledger on the blockchain, the second tier says, well Jeff gets 40%, Derek gets 20%, so-and-so gets 40%, something like that, and their second tier is for each of those main three buckets. Again, it doesn't have to be an ICO, or even a fiat if you just think of we charge for it from a consumption standpoint and then we incentivize operators, publishers and the OSS community. I feel it's something that I had contemplated and we've looked at. We could move to that model as it possibly makes sense. It's just not a requirement for the overall plant.

[1:04:34.2] JM: Yeah. Yeah. You don't need to get an itchy trigger finger any time soon on that one. Well, I'm excited about that. That's a really interesting development and I'm looking forward to covering that more in the future. I'll be at Copenhagen and probably at the next KubeCon. I think that's in Seattle after that, but I'm looking forward to seeing you there and hopefully we can strike up a further conversation about that. Maybe you can get me in on the pre-sale.

[1:04:59.4] DC: That would be great.

[1:05:00.1] JM: I mean, not the pre-sale, because it's not an ICO, but get me in on the pre-sale.

[1:05:04.9] DC: That would be great. Jeff, thank you so much for the time. I really appreciate it.

[1:05:07.8] JM: Thanks, Derek. Great conversation.

[END OF INTERVIEW]

[1:05:12.6] 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 plugins. Use the value stream map to visualize your end-to-end workflow. 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 long-time sponsor of Software Engineering Daily. We're proud to have ThoughtWorks and GoCD as sponsors of the show.

[END]