EPISODE 722

[INTRODUCTION]

[00:00:00] JM: Slack is a real-time messaging system for work communication. On Slack, chat rooms as big as 100,000 people have productive conversations. This might sound like the same problem solved by social networks, like Facebook, where billions of users communicate over a newsfeed. But the engineering constraints of a messaging system are different than that of a social network. On a newsfeed, the order in which events appear is not necessarily chronological. Events can be out of order. You can miss events. When a user posts a message to a social network, there are not strict guarantees around when other people will see that message.

On Slack, messages have strong guarantees around arrival. When I send a message, everyone else who is in the room and connected should quickly receive that message as well. The messages need to be ordered and delivered exactly once. All messages on Slack are persistent.

In the past we've covered architecture and security at Slack. In today's show, Keith Adams returns to discuss how messages are processed and broadcast in Slack. The problem of Slack's messaging system is similar to the distributed systems problem of atomic broadcast, in which is a single process broadcasts a message which needs to be received by all the other processes correctly, or else received by none of the other processes.

In Keith's last show, he talked through the benefits of building a large system on PHP. Keith previously worked on infrastructure at Facebook, which was also a PHP application. It's worth noting that both Slack and Facebook have scaled a monolithic architecture, which runs in slight contrast to the fever around microservice architecture.

[SPONSOR MESSAGE]

[00:01:53] JM: Managed cloud services save developers time and effort. Why would you build your own logging platform, or CMS, or authentication service yourself when a managed tool or

API can solve the problem for you? But how do you find the right services to integrate? How do you learn to stitch them together? How do you manage credentials within your teams or your products?

Manifold makes your life easier by providing a single workflow to organize your services, connect your integrations and share them with your team. You can discover the best services for your projects in the manifold marketplace or bring your own and manage them all in one dashboard. With services covering authentication, messaging, monitoring, CMS and more, Manifold will keep you on the cutting-edge so you can focus on building your project rather than focusing on problems that have already been solved. I'm a fan of Manifold because it pushes the developer to a higher level of abstraction, which I think can be really productive for allowing you to build and leverage your creativity faster.

Once you have the services that you need, you can delivery your configuration to any environment, you can deploy on any cloud, and Manifold is completely free to use. If you head over to manifold.co/sedaily, you will get a coupon code for $10, which you can use to try out any service on the Manifold marketplace. Thanks to Manifold for being a sponsor of Software Engineering Daily, and check out manifold.co/sedaily. Get your $10 credit, shop around, look for cool services that you can use in your next product, or project. There is a lot of stuff there, and $10 can take you a long way to trying a lot of different services. Go to manifold.co/sedaily and shop around for tools to be creative.

Thanks again to Manifold.

[INTERVIEW]

[00:04:08] JM: Keith Adams, you are the chief architect at Slack. Welcome back to Software Engineering Daily.

[00:04:12] KA: Thank you so much, Jeff. Great to be back.

[00:04:14] JM: Yes. The last time we talked, we spoke mostly about the overall architecture. We talked about PHP. We talked about your time at Facebook. It was a great overview. Today I'd like to go deeper into the concept of messaging and particularly how to scale a messaging system and how you've done this at Slack. If we start at a high-level and we're looking at your time in the past couple of years, what has stood out as the biggest challenge in adapting the architecture to the scale that you have reached at Slack?

[00:04:46] KA: That's a great question, and I'm going to whistle out of it initially and sort of say it's been a million little things. But I think I have an insight into the shape of the million little things and what thread unites them that took me a while to form. But it's going to take a little bit of background to kind of zoom in. Starting way out in sort of classical academic distributed systems, one of the first things that you want to do when you string two computers together is build a reliable messaging service between them. As soon as you've got a computer talking to some other computer with a wire between them, you want them to share a log. You realize right away that if I could just have a persistent log that they both could share and agree on the contents of, I could do all kinds of neat things, or I could build a fault tolerant database out of that. I could build a product like Slack out of that. I could build a product like Bitcoin out of that.

It occurs too when you sit down and start coding and trying to make this happen, you find out pretty soon that the fact that these are physical objects in a physical world with failures and delays and possibly even mangled messages, makes this really difficult. You kind of bang your head against the wall for a while and maybe move on to something else. But early distributed systems academics realize pretty quickly that there's actually an impossibility result heading out in here. Around the same time as consensus was sort of shown to be impossible. They also prove that the problem that Slack is sort of approximating a solution to, which is called atomic broadcast, is also impossible.

Let me try and describe sort of in terms of that a Slack user would find familiar the requirements of a channel. What's a channel supposed to do? Well, we have this understanding that when I send a message in a channel, everybody is going to eventually see that message who's subscribed to the channel. We have this understanding too that they're on mobile devices, they're on laptops, they're online, they're offline, but we have a sense that eventually they'll get back online and then they'll see all the messages that have been sent and they'll see them in the same order.

We also have an intuition that if I receive a message, everybody else will eventually receive it as well until the channel sort of keeps the channel abstraction, the basic product channel abstraction guarantees. The channel doesn't hallucinate messages. The only place that messages come from are the fact that people send them. Finally, we got a total order. Everybody agrees on the order that the messages arrived in. It turns out those four things are actually the definition of atomic broadcast. There's a very strong kind of impossibility result at the core of the kind of product that Slack is trying to be.

That doesn't mean we lay down and die of course. Slack exists. It's useful. We strive to make it more useful, but it does mean that there are going to be some corner cases and some failure cases and some contingencies. We will have some hard decisions to make. Those decisions are going to be informed probably by product choices. The way that you want to handle the fact that messages can get lost or reordered is going to be different when you're dealing with something like user presence, the little green dot next to names when people are online. Then you're going to do for text messages. Then you're going to do for files. Then you're going to do for [inaudible 00:07:41] to files and so on.

That combination of there's no one right way to do it. There's a strong negative result that says you can't have everything you'd want from a system like this is going to mean that we keep having interesting challenges; scaling, performance, correctness and making the right tradeoffs when something has to give.

[00:08:00] JM: What you're describing there is – I took a distributed systems class in college and it was really hard, and we implemented some of these algorithms, like atomic broadcast, or Paxos, in a toy environment, where we pretty much understood the failure domains that our application was going to have to satisfy. We didn't have to serve real-world traffic. We were just controlling all of the software. Even then, it was quite hard to understand what was going on. It was quite to understand the basics of the algorithms and what was impossible. What you're saying here is that when you grow up and step out into the real-world and you're no longer in the realm of academic computer science, this stuff still exists.

[00:08:43] KA: Yeah, that's right. Actually, in some ways those are – Some of them are interesting places to be. I think that a lot of great technology companies have at their core some sort of apparent impossibility result that they're running up against and that they insist on making a good run at anyway. So down at the computer science level, Slack's challenge is the fact that it's atomic broadcast and the fact that you're going to have to make choices about exactly what you're going to do about the fact that clients come and go and the fact that servers fail and the fact that storage environments have all the messy characteristics that storage environments do, and you have to do it in a world of 2018 where people's expectations around how well this stuff is going to work are set very, very high. Nobody is interested in hearing Keith explain to them why Slack was slow today. In distributed systems terms, they're just annoyed that it's slow.

[00:09:32] JM: Harkening back to our last conversation, this is an interesting contrast with Facebook where, obviously, Facebook has a messaging product too. But the core product is the newsfeed. I mean, I would argue. The newsfeed is more of a – Not even eventually consistent. You don't even have to time order. It's like throw some stuff in a list and then serve it up to people.

Atomic ordered messages is not at the core of Facebook's problem. That may have been a contrast in the core problem set Facebook to what you're experiencing at Slack.

[00:10:05] KA: Yeah, I think that's absolutely fair to say that's sort of the set of gigantic scaling, consumer-facing solutions that got kind of mastered in the late aughts, early teens by Google and Facebook and Twitter and lots of other companies. That playbook is very focused on a world of sort of information retrieval or whether some ranking function that does better than chance that tells us what we want to see and the idea that there is a ranking function that's deterministic that's forced on us by the product choices and that missing anything is still a wrong answer. We want to have tight latency tolerances and have things feel fast and slick and highly available actually makes Slack of a really challenging environment from a distributed systems point of view.

Since you mentioned some of the consumer-focused messengers, Facebook has a messaging product, as is Google, as is everybody these days. It's one of the things that tends to get tacked on the side of consumer-oriented logins these days. There is a difference in practice between supporting conversations among a handful of users, which is really what those consumer-oriented ones let you support, because they make you compose the two line. You decide who this conversation is about, and maybe if you're having the hairiest conversation you'll ever attempt in Facebook Messenger, you'll try to get 12 people into it or something and you then realize you forgot a 13th person. But you're never going to get to 100,000 people that way. You're never going to get to a huge enterprise. Literally, everybody is going to open their laptop at 9 AM and expect to read what the CEO wrote in the announcement channel, and that's more the kind of challenge that Slack has to get to.

You might say that, "Well, do we want atomic broadcast for something with an audience of 100,000?" you know what you're describing in some way sounds sort of like a WordPress blog. That's actually a fair observation. I mean, part of what makes Slack interesting is that while we've got one product abstraction in the channel, the constant factors between you and I having a DM thread where it's just the two of us on our mobile phones and an entire enterprise looking on their laptops something with 100,000 other people are going to try to read at the same time means that we probably do need to have more than one trick in the toolbox for how we handle this stuff.

[00:12:12] JM: Could you give a few examples of what kinds of patterns work when you have 100 people in a channel versus when you have 100,000 people in a channel.

[00:12:23] KA: Yeah. So with small numbers of people, the tradeoff between sort of pull and push becomes a lot less critical. Imagining the limit that it's just you and me, the difference is between – And let's say that we can tolerate a couple seconds of staleness. The difference is between you and me just pulling some backend asking, "Hey, are there any messages that I need to get every couple of seconds?" That's a pretty acceptable chunk of server load. But if you have gazillions of people trying to do that to the same conversation, first of all, whatever server-side object actually backs the unread state of that is going to be pretty heavily loaded.

Secondly, you'll obviously have that larger factor of people fanning in to do it. So it becomes more critical to actually either have some sort of push story. In our case, we use web sockets as the technology where we use push to get into clients quickly, or to have a carefully considered pulling story. That carefully considered part gets complex quickly.

So let's say that you take that insight and you run with it and you say, "Okay, we're going to do web sockets now." So Slack is going to behave strictly as sort of server push way. There're all kinds of information that is in the Slack client needs to keep up-to-date with that isn't necessarily well-served by a server push model.

A good example of this is imagine user profiles. So Slack users have usernames, they have a status, they have a profile picture. Those things change and the change at sort of a – For most users, a pretty slow pace. It'd be unusual to change your profile picture more than once a day, for instance, in a work product like Slack. If you have this web socket as the only possible way to get updates into a client, you have this temptation and this is a performance problem we had early on at Slack with large teams. You have this temptation to broadcast every possible change out over it.

For instance, let's say that you and I work at 100,000 person company and we're both Slack users at this company. If we simply broadcast every user profile change over the web socket, pretty quickly all the traffic is going to be nothing but user profile changes, even though people don't change their profile more often once a month. This led to a bunch of kind of nasty networking problems in our right amplifier that gets messages out to clients.

In that particular case, the right answer is probably something more like some kind of pub/sub architecture where a client would know what information it has cached or what information it has in view and would only subscribe to real-time updates for the set of things that it actually cares about. But then that means that there's more state associated with your connection. Now, the server side of your connection needs to have some sense of what you're interested in, what it can sensor upstream and this sense of kind of rippling pretty far back into the infrastructure.

We keep finding little ways to improve these things. A big example recently for us was a project that we called Lazy Channels, and this gets to sort of the graph structure that's inherent in a Slack client. It's very hard, and this happens to me too by the way, even though I do this all the time from the engineering side of the companies I've worked at. It's very hard as a user of one these piece of software to not think about it in a single player way. Your experience of it is whatever you see. You tend to think of it as, "Gosh! Slack just has some icons and a few messages that needs to get to me. Why is this so hard?"

In a large company, channel creation tends to more or less linearly follow user addition. The number of active channels at a company tends to be about linear with the company's size, and that's like a sociological client. That turns out to be true. You have to actually build Slack and deploy it and let people use it before you know it. I know of no first principles reason that that should be true. It just turns out to be true. You might imagine, you can make up a story why that shouldn't be true. You could imagine that that's what the – That if the channel structure follows the org structure, it should be like log and the number of users or something. That's not what happens.

That means that the graph that connects users to their subscription of channels, so which users are in which channel. That means that that graph grows quadratically, because every user comes along. They create some fraction of a channel. Now, the binary matrix that says, "Is this user in this channel is num users by num users?" If, as we did when Slack was young, we tried to synchronously pump that matrix down into every client as soon as you connect, you're only going to be able to support certain sizes of teams.

We have this system that we call Lazy Channels now that basically applies this sort of pub/sub architecture that's a lot easier to think about in terms of user profiles, but applies that pub/sub architecture to channel membership. So each client is only getting information about the channels it cares about at any given point in time. That's been a big win for us in terms of reliability and startup time on large teams recently. But stuff like this is going to keep coming.

[00:17:11] JM: Lazy, referring to lazy loading. Meaning, when I connect to Slack for the first time or let's say I get a new phone and I connect to Slack and my team is already there with its tens of thousands of messages and thousands of users. If I hit Slack and my phone says to Slack, "Hey, give me all of the messages that are relevant to me so that I have a backlog that I can look through and then I can scroll through and they can understand the relationships of everybody." It's a lot of data. You're not going to be able to just download all of that on-the-fly if you've had a team with all of these backlog of stuff. You've implemented this lazy loading system so that your channel information kind of comes in at a pace that's reasonable.

[00:17:58] KA: Yeah, precisely. I’m using [inaudible 00:18:01] in sort of the classic big system design sense, where the nice thing about doing work eagerly is if you know you need to do it, you get started as soon as you can. But the nice thing about doing work lazily is it turns out you don't need to do it all all the time. That's exactly kind of one of the dichotomies that keeps sliding around for different parts of the product. I think it's a function of how people are using the product in certain ways. There's no kind of right way to design it. When you hear me give this talk, you’re like, “Well, yeah. Of course, you’d want to do channels lazily,” but you kind of need to see how people turn out to use Slack before you know for sure that Lazy is the way to go with it.

We kind of went deep into this example of channel membership, but there's a million little objects inside of Slack that are kind of like this. Files are like this. Custom emoji are like this. We talked about users. The sort of existence of channels, whether they’ve been archived, things like that. It radiates outwards in lots of different ways. It's a little bit – This sort of eager versus lazy dichotomy, I think it helps to kind of know little bit about the shaggy dog story of where Slack came from for why it started out so eager. There is real technical DNA for Slack’s broadcast message bus that dates all the way back to it being a game server for the massively online game glitch, which is what the company Tiny Speck was originally formed to produce.

Slack, the messaging product, is partly a piece dividend of that failed massive multiplayer online game that they came together to build. In the game world, what's going on when you’re looking at that splash screen with all the ogres on it and so on, is that you’re getting a somewhat consistent view of the whole game state on the server so that every time you turn around the corner, there's not some glitch where something pops into view while you go fetch it over the network. You want a pretty coherent view of the game world so that you have a sense of presence so that it feels like a virtual place.

To the extent that this is knowable, I think that sense of presence and virtual place is one of Slack's intangible advantages in this space. I think that is one of the reasons that Slack tends to feel more like an office than like a program that you type text into and that spits text back at you. I can't prove that and I don't know of any easy way to run the experiment. But that's one of the reasons we’re not totally reckless about just saying lazy everything all the time.

[00:20:21] JM: It's such a useful analogy, because you mentioned this term graph, like this Slack's idea of – Or I think you said in terms of the client, you could think of the client's data in terms of a graph. When I think about a game, like if I'm playing an online game like World of Warcraft and there's all these data that's going on in the world, I'm not downloading all of the data. I'm incrementally getting kind of the data that's local to me, that around me in my virtual environment and then some horizon around which I might explore so that I'm not going to hit the edge of the world that's sitting on my machine anytime soon. I'm always getting the edges of the world buffered to me. It seems like at Slack you kind of got the same thing. I mean, you want to be able to click into any channel and have some context for what's going on in the channel, but you don't need to have the entire state of the world eagerly buffered to your client.

[00:21:17] KA: Precisely. I think there's an interesting – And games are fascinating technical artifacts by the way. I mean, I think we should – Those of us who don't make games, it's wonderful to learn a little bit about how games work on the internal side of things. I think though the interesting difference between sort of Slack and something like a virtual game world is that there is no spatial analogy with Slack. In the game world, there's a clear notion spatial locality. They've tiled the universe in some spatial way. Even if I'm running around it very quickly, there's some direction that I'm running it in and there's only a set of neighbors that I could possibly go to next.

Whereas somebody consuming their channels, like maybe you could machine learn something predictive. But all by itself, there's a chance that they'll kind of randomly access their channels. It's hard to kind of get ahead of it in the same way that you can in a game world.

[SPONSOR MESSAGE]

[00:22:12] JM: A thank you to our sponsor, Datadog, a cloud monitoring platform bringing full visibility to dynamic infrastructure and applications. Create beautiful dashboards, set powerful machine learning-based alerts and collaborate with your team to resolve performance issues. You can start a free trial today and get a free t-shirt from Datadog by going to softwareengineeringdaily.com/datadog.

Datadog integrates seamlessly with more than 200 technologies, including Google cloud platform, AWS, Docker, PagerDuty and Slack. With fast installation and setup plus APIs and open-source libraries for custom instrumentation, Datadog makes it easy for teams to monitor every layer of their stack in one place. But don't take our word for it, you can start a free trial today and data dog will send you a free t-shirt. Visit softwareengineeringdaily.com/datadog to get started.

Thank you to Datadog.

[INTERVIEW CONTINUED]

[00:23:20] JM: Since we're on the subject of games, there was a show that we did fairly recently about network connections in gaming. As an example, let's say you're playing a two-person fighting game, even just a two-person game, and we're playing over a cellular network. So it's kind of flaky. Let's say I throw a punch at you in the virtual game and then there's like a network partition, but in your view of the world, like maybe you're on a better network than I am and your view of the world is that I have not thrown a punch at you yet. So you're going to throw a kick and then there's kind of a conflict resolution that has to happen because we're not punching and kicking each other at the same time.

So on the server, when the server reconciles that, the server is going to have to make almost a subjective decision around what came first, the punch or the kick? From the talks I've seen you give about Slack, there is some subjectivity here on the messaging side of things too, because you have to do the ordering of messages, and the ordering of messages can be analogous to who punched first or who kicked first.

[00:24:28] KA: Yeah. It's exactly true. There are a few interesting cases around typing in the tale of a channel, checking the last message in a channel, where one of the things that we've chosen to get "wrong" from the point of view of trying to solve atomic broadcast is it is possible in some circumstances for me to see my message sent first and then you spoke and you to see the opposite order. It turns out that's only for a short period of time and only for the last few messages in a channel. That's something where the other things we'd have to give up on to make that impossible worth it, it would have involved things like doing more round trips back and forth to server before we could give you the feedback that says your message is sent, which is one of the things that make Slack feel subjectively sort of correct.

There's also – With the particular example of message sending, we've kind of gone through a revolution of that over the last couple of months in a major way. One of our founders is Seguei Mourachov. He's the tech lead still of our message bus team, and a great programmer, fun we got to work with. One of the things that has historically been unusual about Slack as sort of a web socket client has been that you can actually mutate the state of the application by writing up the web socket.

For those of you that haven't used them before, web sockets are connection-oriented and they're full-duplex. So it's not quite the same thing as an HTTP long pole. You're free to write framed messages up to the server and the server is free to send framed messages down to you. For a long time, we actually would do message send by transmitting a message up to this real-time message bus, which then on the backend was responsible for persisting it. The story around how it actually got to be persistent and stable storage in the system that – The system of record for us, which is MySQL, was slightly convoluted. That Java program that you're writing to doesn't synchronously contact MySQL or even a web app that contacts MySQL while you're waiting for feedback. It could actually lead to some unusual and kind of strange cases.

For instance, if the storage system or web app in front of the storage system was backed up, there was actually provision in that Java program to queue some messages on disk and retry transmitting them back into storage. These are all sort of messages that we've act to users as being fully sent, as being fully backed, and part of those stream of record.

Slightly convoluted case here, but long story short, we now post to a REST endpoint, the same API endpoint we advocate third parties used to send messages from our first party clients, because that was easier to reason about. But one of the things that we had to give up on is this guarantee of ordering. It used to be that that Java server on the opposite side of your channel connection was the authoritative witness of the order of messages and could never reorder anything. Now it's possible for clients to have sent something at essentially the same time from the perspective. So the way that it might appear in your client view might be different than the way that it appears in the store of record and you'd go back tomorrow and see it in the right order.

Luckily, Slack is for human beings, and human beings are able to make sense of these distinctions. It's one of the areas where you kind of get to relax atomic broadcast a little bit as you sort of say, "What uses are people actually putting this software to? Are they trying to build a distributed database out of this? Are they trying to build a distributed ledger out of this?" No, or they shouldn't be anyway. It's not a good technology for it.

[00:27:55] JM: Not yet.

[00:27:57] KA: Well, yeah. I think it's probably out of scope for us. I think we want to keep this humans are in the loop in channels capacity as Slack as a way of settling bets on the technical architecture for a long time if we can.

[00:28:09] JM: I wonder if the Slack chain ICO would get some buyers.

[00:28:16] KA: Oh my goodness! Yeah, I think some sort of bingo card just got [inaudible 00:28:18].

[00:28:19] JM: Exactly. Right.

[00:28:23] KA: Yeah. I suppose I technically shouldn't comment. But, yeah, I wouldn't be holding my breath for that anytime soon.

[00:28:28] JM: You started touching on the architecture of the chat system, and I wanted to review the architecture, because I was looking into it yesterday and it's kind of complicated. But I think we can approach it in a way that will be useful enough to the listeners. So if I send a message from my phone and I'm hitting Slack, that message is going to go through some layers of routing and load-balancing, but eventually it's going to hit something called a chat server. Then you have a web app. So the chat server is kind of this dedicated thing on Slack side of things that manages some of the chat message ordering and some other aspects of the chat world. Then the web app is sort of like the back and the backend. Can you give a brief overview of these two pieces of infrastructure? Then we can talk about them in more detail.

[00:29:19] KA: Yeah, absolutely. At the risk of I'm picking a knit with, of course, we're not going to be able to go into all the detail in the world here. But for what it's worth, sending a message these days does actually go through the web app that I'm about to describe.

When you go to slack.com, there's a web application on the other side of that and it goes through, as you're saying, a fairly typical through a global scale service layer of DNS and routing and everything else until it makes its way into our application tier. One of our application server is a descendent of a LAMP stack app. There's a bunch of sidecar is running. There's a bunch of other stuff running, but from 100,000 feet, you can tell that this used to be a PHP monolith talking to MySQL.

MySQL is still in the room. The PHP monolith we've migrated it to Facebook's eventual typing system, our gradual typing system for PHP called Hack lang. So we're using HHVM, which is the Hack lang virtual machine instead of PHP on these servers. But the thing that's actually terminating HTTPS for you is Apache, and Apache is talking over a fast CGI, about a million lines of PHP code that we've written. That monolith is sort of the authority about application logic in Slack. It's what knows what a channel is. It's what knows what a user is. It knows the rules about what kinds of users can create what kinds of channels and who can invite whom and who can see what objects. There's a lot of Social Security reasoning that goes into this. It does authentication and it is also the sole contact integration point for storage for a database.

We are partway through a migration from vanilla MySQL sort of sharded by the client, where the PHP actually uses application logic to figure out which MySQL server to contact for a given data to a system called vTEST that was originally developed at YouTube. VTEST is a layer of scale out on top of MySQL. So instead of sharding MySQL in client-side code, we have a certain meta-schema for how we want the test to shard our data. The test then talks to a bunch of leaves that are running MySQL as well.

It's still physically MySQL. MySQL still actually got the bits on disk and still a thing that we backup. We also have a bunch of caching. So we're using mem-cache to look aside cache. We use Redis as a buffer for our job queue deferral system. We have a e system for different work to later in time in case a web request is running too long. We call that the job queue. It's not very creative, but lots of places have fancier codenames for their setup.

Basically, we have a big set of Kafka topics that we pump deferred work into. We have a little Go service the polls those Kafka topics and buffers them into Redis. Then we've got a fleet of workers also running a PHP monolith consuming out of the buffered jobs in Redis. This is how things like unfurls when you type a URL into Slack and we make a little box around a summary of it. We might need to retry it a few times, because that website might be down when you typed it the first time and it might be long-running. That website might just be slow. It might just take us 30 seconds to unfurl it. You want to take that out of the context of the web request that you did the initial send in. We do similar things for search indexing, for lots of other kinds of value addition to minor application action. But this part of things really is your father's web app. This sort of thing is understandable in terms of our CTO, Cal Henderson, was well-known for scaling Flickr before he got started in what would become Slack. He wrote an O'Reilly book that's titled *Building Scalable Websites*. If you're going to read that O'Reilly book, there'll be a bunch of stuff that sort of is obviously in the DNA of the application still.

[00:33:03] JM: So the channel server though, when I am like sending a message from my phone. After I've established – I've initialized my connection with Slack. I've logged in. I'm already sending messages. I'm connected. If I send the message, is that message being hit immediately by channel server after it goes to the layers of load-balancing, or is any individual message like hitting the web app as the first tier?

[00:33:31] KA: Nowadays it is the latter. But as recently as a couple of weeks ago, I think there probably still were some people [inaudible 00:33:36] that group where you'd have written to a channel server to send the message.

Let's switch over the channel server here for a second. Let's say that this is all the technology I've described so far, which basically is just a big website. Let's say that you want to build Slack and you're only allowed to use those kinds of technologies. Without the element of server push, you're going to be forced to use the kind of pulling methodologies that I referred to early on in our conversation here, where if I'm a client of this Slack service and I have no way of being notified asynchronously when something's changed server side, I'm just going to have to pull. I'll have no choice but to say, "Any new messages? Any new messages? Any new messages?" on some cadence, right? That turns out not to be a very efficient use of client or server resources, and you don't get very good latency for the effort. Plus it's hard to furnish all kinds of nice little features that make it feel more like a virtual place. It's really hard to do typing indicators that way, and typing indicators really help a lot with message services. Help humans coordinate their conversation.

We have this system and we're going to use the channel server as sort of a representative for this, but the reality is it's kind of a constellation of services that runs in lots of different places now. The goal of this constellation of services is to provide a real-time channel abstraction. To provide something that you can long pull on. Something that you can select a few old-school UNIX terminology. Clients consume from this pipe by opening a web socket to a URL that they get from the web application. The web application as part of your login process, it looks at your token and says, "Oh, okay. Yeah, you're Jeff Mayerson. You're part of these teams. Here's a URL for you to go listen to," and you go listen to it.

Then it is the responsibility of this server infrastructure to make sure that updates that real-time clients care about get broadcast over that bus. There are side calls sprinkled throughout the web application into this real-time service to say things like, "Jeff started typing," or in the case of message send, "Hey, a message got sent. Let's broadcast it to all the people that care."

When somebody logs in or opens their laptop or focuses the application for the first time in a while, it will say, "Hey, user so-and-so is present again," and clients might stick a green dot next to your name in response. So that real-time part of things, and in a perfect world, the bus is kind of stateless. We should be able to power cycle the whole thing and all that would happen is that people aren't getting updates for a little while. That's an ideal we haven't lived up to very, very well until recently in my opinion. Conceptually, that real-time part of things, we try to keep that a sort of separation of church and state between the stateful part of things that we backup and involves application level reasoning and that we can make guarantees to customers about when they have regulatory compliance demands and things like that. The ephemeral only in memory long enough to be transmitted over the network part of things, which is kind of represented here by the channel server.

The reason they're called channel servers is that the semantic level, the domain of ordering inside of Slack is the channel. There is no global ordering of events that is not within a single channel. For instance, if you look at your Slack team and it has 700 channels and there are timestamps that represent server timestamps attached to those messages, but there's no guarantee across those channels exactly what order the events happened. If something has the same microsecond timestamp, there's no way to say which [inaudible 00:36:59] before.

For messages and other stateful events, like the creation of threads and sharing files and things like that, within a channel, there is a total order. The way that we guarantee that total order is that the channel servers themselves are a consistently hashed ring by channel ID. So when I want to send a message, I take the idea of the channel I'd like to send it to, I project it on to this ring of servers. Last I checked, it was in like 40 servers. That's the server that I'm going to go hit to send the message in. Its job and privilege and responsibility to provide a total order on all the messages. It the thing that actually gets to decide what goes out in what order over the web socket to clients.

[00:37:41] JM: The channel server is, if I understand correctly, is responsible for imposing this ordering on messaging. It's responsible for receiving messages and then broadcasting those messages to other users that are in the same channel, the same channel on Slack. I believe there's also a responsibility of check pointing those messages to disk so that if it in case the channel server falls over, you have some redundancy.

[00:38:12] KA: So that was true at one time, and we no longer do that as of relatively recently. One of our quarterly goals was to get the on disk queue of the channel server out of the loop. As I sit here talking to you, I believe that's finally completely true and we'd have to get Serguei or another engineer on his team in here to the make sure I'm not lying to you. But if that's not true yet, it's awful close to true.

[00:38:33] JM: Was that an anti-pattern? Because then you were dividing up the persistence responsibilities between the channel server and the web app. Like the web app was kind of thing that handled check pointing things to your MySQL source of truth, and you had this other kind of persistence backup thing.

[00:38:53] KA: Yeah. I mean, anti-pattern might be a little bit strong. I think there are perfectly good reasons to done things the way that they were done initially, I think, but it was a recurring source of some complexity, especially as we do more and more products where we make representations to customers about where their messages are or how we're handling them.

For example, my colleague, Richard Crowley, gave a talk at Frontiers, which is our user conference recently, about an EKM product we're rolling out pretty soon hopefully. EKM stands for enterprise key management. The idea here is that we have customers who, for their security sensitivity reasons or for compliance reasons, want to hold the encryption keys to their data. They don't want Slack to be able to read their messages even in principle. They'd like to be able to revoke Slack's access to all their data at some point in the future if they need to.

So we have a system that we're – A product that we're trying to put together that lets you take your key management system and integrate it with your installation of Slack, such that Slack uses your keys to encrypt and decrypt data. A little bit of thought of – If I can kind of brag about this for a while, like Richard kind of came up with something really pretty here, because it's really hard to do this and preserve search, and preserve lots of other value ads inside of Slack, and we think we're going to be able to do it. But that's an example of a product where when we're sitting there trying to plan it out and figure out how we're going to do it, we kind of kept coming back to that little on disk queue of messages on channel server and saying, "Oh! Right." Those ones, "Does the channel server need to be able to talk to the key management system now? Do we want that to be in the loop for a send?" It just sort of felt like a big wart on the side of the system.

Sort of similar stories around some data loss prevention products that we have. Also, we have retention settings. So some people, we tell you we'll delete your data after seven days, or 24 hour, or whatever they want. It was another thing that we had to make sure was sort of looped into this that was on a completely different system, that wasn't part of MySQL. It also was something that was a fuse that was lit if sort of the site was down. Imagine that sort of slack.com is unable to process web requests for whatever reason, but people are still logged in and sending messages on channel server. If we've acknowledge receive of those messages, we sort of got this on disk queue growing in a tiered machines that is a sort of lit fuse that will eventually run out of disk space. That limits our time to be able to respond to the outage. Yeah, it was something that I think caused a lot of complexity. I'm sure the way that we're doing it now has and will also continue to cause problems. We just like those problems better.

[00:41:26] JM: Fair enough. One question I had about the channel server. I think I have – No, I have not done a show with WhatsApp. But WhatsApp, they use Erlang, and I've talked to other people that use Erlang for managing these kinds of multicast chat systems. You use Java for your channel server. What are the advantages of using Java, and have you looked at Erlang at all as a potential use case? Because I hear Erlang mentioned a lot in the context of high uptime chat-like multicast systems.

[00:42:00] KA: Yeah. Actually, I should say before I go any further here that we're not a Java monocultural shop. Our calls team actually uses Elixir for its backend. So the video chat and screen sharing and voice calls inside of Slack. Elixir is in the loop there. Elixir is kind of modernization and syntactic sugaring on top of Erlang. To be fair, they seem to be sort of falling into a pit of success using Elixir there. So that's great. We may use more of it in the future.

I think that the choice of sort of tooling on the part of the founders, including LAMP stack, including Java, including the fact that the client is sort of framework was JavaScript. At least classic Slack's web application was JavaScript without Angular or React or anything, just JavaScript. I think a lot of that had to do with just what our founders were most productive with.

[00:42:49] JM: Nothing wrong with that.

[00:42:50] KA: Yeah. They weren't interested in necessarily learning something new. They knew exactly how to do this with the tools that they knew best, and so they just got started doing it. Java has not been the weak link in the chain for us for what it's worth. The sort of first-class service languages at Slack that has some high-volume adoption inside of our services are Go, Java and Hack lang. We do have some services that we've all [inaudible 00:43:12] in Hack lang. Not just the web service. Elixir is a minority choice. Calls also have some C++ services. As we make some acquisitions, we're starting to acquire some small startups. We also end up integrating their stacks. They bring in some diversity in that realm as well.

We’re not kind of perfect monocultural Java, PHP or Go home type of place to date. Also, I think it all kind of fits into the pragmatism that informs the sort of technical spirit of Slack. It’s a place that kind of flies in the face of an impossibility result and does something useful anyway. So kind of culturally and spiritually, we’re not the kind of place we’re going to say, “Right! We [inaudible 00:43:53] with Erlang.”

Since you mentioned WhatsUp by the way, and this is not – I was briefly at Facebook after WhatsApp acquisition, but never worked on WhatsApp. But my understanding from public stuff about WhatsApp is that at least classic WhatsApp, sort of cellphone to cellphone, small number of participants, was actually in-memory server-side entirely. There was no actual persistence to it. The messages were actually only stored on phones. They were queued in memory server-side until you started the WhatsApp client.

That's not to say that you can't ever do persistence using Erlang, but that seems to be a little bit of a unifying thread for the things where Erlang is extremely successful. With us with calls, that Elixir service on describing calls also has that property right. We’re not recording your audio or your video for you. It seems to be a little bit of a unifying thread for the use cases for Elixir, whether super, super successful. Not to say you couldn’t use it in other places, but that’s a thing about WhatsApp that I think was actually really freeing for me to get and sort of architect at Slack, was the WhatsApp is providing this great performance experience on super low-end devices, like why is Slack having so much trouble? Well, part of why Slack is having so much trouble getting its performance is actually that it's doing things that it's not doing, that WhatsApp wasn't doing. It’s also why it was so easy for them to do end-to-end encryption. They weren’t actually giving up any data at rest to do that. They weren’t storing your messages anyway.

[00:45:13] JM: Yeah, sure. On Slack, in contrast, every message in every channel is persisted. That is a lot of messages. What is the spec for persistence of messages in Slack and how have you implemented it?

[00:45:29] KA: That's a great question. Our store of record, as I mentioned before, is MySQL. In some cases fronted by the tests and sometimes directly accessed as MySQL. In terms of how we physically are persisting these things, it's a table on MySQL named messages. Your message is uniquely identified in the world by a channel ID and a timestamp. The timestamp is fully ordered by channel servers. We get this timestamp out of channels servers. That's where the uniqueness guarantee comes from.

This is kind of an interesting thing, but since your use of Slack is dominated by – Or many people's use of Slack getting away is dominated by the sending and receiving of messages. Since so much of kind of what we're talking about here is about the real-time aspects of that, you might expect that to be really storage-intensive. In practice, the storage-intensive parts are actually different parts of the application.

We spent a lot more bites, at least last I checked, representing things like channel membership than we do representing messages, which might feel a little strange. But if kind of go back to that fact that I can only send somebody messages per day, as the company gets bigger, more and more people are going to join more and more channels. So that channel membership graph has kind of grown quadratically with customer size, while the number of messages sent is growing kind of linearly with the member users we had.

In practice, this is not where we start to get into trouble with burning up MySQL servers. It's just raw sending of messages. It's much more common that we – In terms of actual performance problems we hit, or storage difficulties that we hit, it's much, much more common that we're doing the same complicated join over and over again trying to tell you many unread messages you have for instance.

[SPONSOR MESSAGE]

[00:47:20] JM: This podcast is brought to you by wix.com. Build your website quickly with Wix. Wix code unites design features with advanced code capabilities, so you can build data-driven websites and professional web apps very quickly. You can store and manage unlimited data, you can create hundreds of dynamic pages, you can add repeating layouts, make custom forms, call external APIs and take full control of your sites functionality using Wix Code APIs and your own JavaScript. You don't need HTML or CSS.

With Wix codes, built-in database and IDE, you've got one click deployment that instantly updates all the content on your site and everything is SEO friendly. What about security and hosting and maintenance? Wix has you covered, so you can spend more time focusing on yourself and your clients.

If you're not a developer, it's not a problem. There's plenty that you can do without writing a lot of code, although of course if you are a developer, then you can do much more. You can explore all the resources on the Wix Code's site to learn more about web development wherever you are in your developer career. You can discover video tutorials, articles, code snippets, API references and a lively forum where you can get advanced tips from Wix Code experts.

Check it out for yourself at wicks.com/sed. That's wix.com/sed. You can get 10% off your premium plan while developing a website quickly for the web. To get that 10% off the premium plan and support Software Engineering Daily, go to wix.com/sed and see what you can do with Wix Code today.

[INTERVIEW CONTINUED]

[00:49:19] JM: In thinking about the overall architecture as I understand at this point. So I send a message from my phone to Slack and it hits layers of load-balancing and routing and then it hits – I guess it hits the web app and then it hits the channel server, and then there's also – At some point it's getting – That message is getting written to the database. At some point the message is getting broadcast to other people who are in the same channel as me. So once the message is acknowledged on Slack's side as being received, then it can get broadcast to the rest of the people.

There is also a point at which if it's the message is doing something, like if I'm sending a URL over the message, then something like URL unfurling. These sort of side tasks that might be associated with me sending a message to Slack. That's handled by this asynchronous job queue that handles the kind of other things that might be associated with the message that is not core to just sending a message that has to be broadcast to everybody else and is written to the database. I guess I'm just trying to understand the finer points of that workflow of a message being sent hitting Slack's servers, getting persisted and getting broadcast back out to the other people in the channel.

[00:50:38] KA: Yeah. I mean, you've got the rough shape of it, and I'll admit that every time I talk about this even with the engineers who wrote it. It takes us a little while with the whiteboard to really convince ourselves that we got the timeline exactly right. With the caveat that it's possible I'm getting pieces of this wrong. My understanding of that post endpoint, chat.postmessage, is that it contacts the channel server with the message. Gets a successful return from the channel server that includes a timestamp. At that point, parallel execution ensues. The channel server starts broadcasting the message to online clients. The web app takes that channel timestamp tuple, and the payload – Not the payload of the message. It writes the [inaudible 00:51:21] of MySQL that I described before. It writes to the messages tables saying, "In channel such and such, user ID thus and such with the timestamp/ID of this wrote this text," and also in queues a drop queue job to do sort of any, at the very least, update search. But possibly some of the key jobs.

Similar, cascading from this is things like I mentioned. I can say @username for instance and it's supposed to notify you on your phone if you are away from Slack. That message broadcasting, that notification sending that happens because of mentions, that all happens outside of the loop of the web request via that drop key system.

Let's say that I've got my nitpicking distributed systems have on and I've just heard this description and I want to say, "Aha! Got you. Keith, what happens if you have sent the message to a channel server, it started broadcasting out to clients. People are seeing it on their phones. People are seeing it in their web browsers and that web request crashes or that web server gets decommissioned before you can go off and persist it? What then?"

What would happen then is that you'd have an inconsistent state. You'd have a sort of ephemeral message that shows up in the state of some clients that is not reflected in the core application. That never gets indexed in search. If you come back in a month and try to look at the channel, it will look as if that message never happened. We're getting very close to sort of the core of the atomic broadcast impossibility results here. You can paper over that, and my understanding of the way things work right now is actually simplifying the situation in a way that's substantial here.

I don't think we actually quite would fail. I believe that we actually persist the kind of proto-message before we contact the channel server and that a cleaner process would actually come back and repair it. However, the core observation that if exactly the right failure happens at exactly the right moment, some sort of [inaudible 00:53:12] of the application is violated is actually true. The certifying part here is that the art at some level is to accept the impossibility result, accept that if something fails at exactly the right place, something terrible could happen, but reason through the consequences of it in a way that results in something useful.

In the case of Slack, sort of strangely enough, one of the things that is most unacceptable to us is a message that should've been deleted, but isn't. In this case, if I'm the sender of this message, my client would have known that the send failed, because the web request failed in this thought experiment. We unplugged the web server or something. So I got some kind of failure response from my attempt to post to slack.com in this case. Assuming the client actually stays up, in practice it's going to try to send it again, because we build the clients that way. It's going to try to send again with a hidden client cookie that's random that each client generates per message and that he will be enough for the server side to de-duplicate and kind of fix this scenario.

Even in situations where things are crashing, or losing connections, and so on, a human interpretable situation ensues. If computers were trying to make sense of this log of messages, it's possible they'd see a message get renumbered. They'd see it sort of appear with one timestamp, and then later on appear with a different timestamp, and that might confuse them. But if these are people with their clients open trying to understand the train of a conversation, they'll be able to do it.

[00:54:43] JM: So much of this conversation has been in the weeds, and that's good, because that's what people listening to this like to hear, and they like to understand. But one thing I think about is, I've had conversations that are reminiscent of this with Uber and Lyft, because those are other applications where there's a lot of real-time sensitivity and you have a multiuser real-time application that is, as we've discussed, in many ways like a game. But my sense is that this stuff is still so hard to build. You have things like Firebase, for example, which I love Firebase, and kind of enable some real-time applications. But my sense is that this stuff is, for most engineers, it's kind of out of reach. You have to have a big company. In order to build real-time applications at scale like this, you're going to have to solve some really hard problems.

Are there any like AWS services that you wish existed? Is there any piece of architectural blackbox blob that you wish could solve problems to you, or do you feel like these are just problems that are inherent in software architecture that you can't turn into some kind of service to just take care of for the average developer?

[00:56:09] KA: I mean, that so really wonderful question. I think it's tough to give a crisp answer. Slack was getting started four years ago. Public cloud has been maturing really, really quickly since then. I think there are actually a lot of PaaS offerings that would be valuable in building something like Slack that if we were starting over now, we probably avail ourselves of.

There's a class of distributed state problems that we solve by scratch with the pieces I've been describing that Firebase could partially solve for us. Things like user profiles and maybe the metadata structure of a team. I feel like Firebase would actually be a great enabling technology for that kind of thing, and if I were starting a Slack clone from scratch today, I'd be really glad that Firebase is out there.

Even building a backend of stuff, there are things that we did by hand in constructing Slack that might make more sense to use a platform as a service offering instead. So the job key that I was describing for you where we go into Kafka and then we [inaudible 00:57:05] Go that takes this stuff out of Kafka and puts it in Redis and blah-blah-blah. If I were starting from scratch today, maybe that would just be SQS or Kinesis or the moral equivalent of those things.

There are certainly important and useful components that have come along. However, I do think when you boil it down far enough, at least in the case of Slack, and it'd be an interesting thought experiment from you to have this conversation with colleagues at Uber and Lyft. But for the case that I thought about more deeply, I do think there're and end-to-end considerations.

For example, this kind of low-level fiddly bits about exactly what happens when something fails, because something has to fail. You have to have some set of semantics that kick in. When a machine fails or when you lose a network connection and so on. That tends to be application level reasoning. Sort of our answer around what we do when something goes wrong in the message send process, takes into account the fact that we have an EKM offering, and the fact that we've got these other plans for how you want the storage backend to work and son on. It's very hard for me to imagine something that didn't make so many decisions for you upfront that it essentially over-determined the kinds of applications you could build. It's hard for me to imagine sort of boiling away the hard part into a reusable service, because my intuition at least is the decisions that Slack made would be the wrong decisions for Uber, would be the wrong decisions for Airbnb, would be the wrong decisions for Google Hangouts.

That's my best guess at this, is that there is a set of things that certainly are productivity enhancing and take away some of the toil of dealing with everything at the level of like IP addresses and code you wrote that listens on a port. But that there are still some core value that the application actually ads in terms of considering all these cases and making the hard decisions.

[00:58:54] JM: Agreed, and Firebase probably works for the 100 user types of use cases. I guess these problems only really become challenging when you have the kind of traction and scale where you can actually hire a big engineering team and have them build these kinds of solutions.

[00:59:11] KA: Yeah, that's right, Jeff. Scale is another kind of choice. Scale is a set of use cases that you choose to support. There is no way for somebody that doesn't kind of have your economic best interests at heart to understand which choices are going to make sense for you.

I recently gave a talk about Scale and Slack at Denver Startup Week, and it was on the topic of scalability. One of the things I tried to emphasize when I talk about scalability is that it doesn't make sense to talk about scale unless you say which dimension you're scaling in. Nobody wants to put or least as far as I know, you can't put terabyte messages in Slack. That's not a dimension we chose to scale it, message sought. Just not valuable for us to go in that direction. There's all kinds of weird decisions differently if that was an important way to scale. It turns out not to be. It turns out a number of users in a channel is. I think whatever kind of abstraction of a channel that I tried to build and make us PaaS offering, there's no way it would get all of the decisions right about which dimensions you might possibly have to scale in.

[01:00:09] JM: Okay. I think that’s a great place to close off. Keith, thanks for coming back on the show. Really great talking to you. Always a pleasure to see the case study of Slack.

[01:00:18] KA: Always a pleasure talking to you, Jeff. Thanks to you and your audience for your time.

[END OF INTERVIEW]

[01:00:24] JM: GoCD is a continuous delivery tool created by ThoughtWorks. It's open source and free to use, and GoCD has all the features you need for continuous delivery. Model your deployment pipelines without installing any plug-ins. Use the value stream map to visualize your end-to-end workflow, and if you use Kubernetes, GoCD is a natural fit to add continuous delivery to your project.

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

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

[END]