EPISODE 477

[INTRODUCTION]

[0:00:00.3] JM: The internet was designed as a decentralized system. Theoretically, if Alice wants to send an e-mail to Bob, she can setup an e-mail client on her computer and send that e-mail to Bob's e-mail server on his computer.

In reality, very few people run their own e-mail servers. We all send our e-mails to centralized services like Gmail, and we connect to those centralized services using our own client, which is usually a browser on our laptop, or a mobile application on our smartphone.

Gmail is popular because nobody wants to run their own e-mail server. It's too much work. With Gmail, our e-mails are centralized. But with centralization comes convenience. Similar centralization happen with online payments. If Alice wants to send $5 to Bob, she needs to go through the centralized banking infrastructure.

Alice tells her bank to send $5 from her bank account to Bob's bank account. This is not how it works in the physical world. If Alice wants to pay cash to Bob, she doesn't have to go and meet him at a physical bank. She just takes out the $5 bill from her wallet and hands it to him. That's cash.

The invention of Bitcoin proved that digital wallets and peer-to-peer payments are possible. But running your own wallet is like running your own e-mail server. It's inconvenient. So we trade decentralization for convenience once again. We use services like Coinbase, where users buy and sell cryptocurrencies in a centralized provider.

There are people in the cryptocurrency community who hate the idea of Coinbase. These people keep their cryptocurrencies spread out on their hardware wallets. Some of these people probably also run their own e-mail servers.

Are these people just adding unnecessary inconvenience to their lives for no reason? No. These are smart successful people. They don't like to waste time. What are they doing running their own e-mail servers?

Distributed systems theory teaches the risk of a centralized computer system. If you have a single server that all of your communication has to be routed through, your computer network will stop functioning if that single server dies.

Today, civilization is reliant on centralized computer systems, and this is fundamentally dangerous. The 2008 financial crisis proved how risky it is to centralize all of our money in the hands of a few people.

The Equifax breach proved how risky it is to centralize our identity in the hands of a few people. What if happens if Dropbox runs out of money and has to shut down? What happens if all of the data centers at Amazon web services get simultaneously wiped? What happens if Coinbase gets hacked and every user at Coinbase loses all of their money?

We have seen centralized systems collapse. The people who are running their own e-mail servers are not crazy. Even if Gmail disappears tomorrow, those people will still have access to their e-mails.

With the example of e-mail, we see that deploying and managing a centralized system is possible. You still can deploy and run your own e-mail server. Decentralization is a desirable feature of computer systems. How can we make more of our applications decentralized?

The cypherpunks spent decades thinking about how to make decentralized money a reality. Satoshi Nakamoto invented the Blockchain, and now we have a computer science construct that enables decentralized money. The Blockchain also happens to enable many other decentralized applications.

By solving a specific problem, Satoshi came up with a general solution. This is how progress often happens in computer science. In order to fix a very specific system, we create a new tool.

That tool can be applied to other systems that we don't anticipate. The Blockchain is a tool that solves one set of problems in a distributed system.

Conflict-free Replicated Data Types are another type of tool. Conflict-free Replicated Data Types or CRDTs for short are objects that can be mutated by multiple users at the same time without creating data corruption.

The most common example of a Conflict-Free Replicated Data Type is the shopping cart. Let's say Alice and Bob share an account on an e-commerce website. Alice is building a house and she wants to buy some tools online. Alice has a shopping cart with a hammer in it. Bob logs on to the e-commerce website from a different computer at the same time that Alice is logged on.

Bob just wants to buy a tuxedo. He doesn't know why Alice left a hammer in the shopping cart and he doesn't want to buy it, so he clicks a button to remove all of the items in the shopping cart. At the exact same moment, Alice clicks from her computer to add a drill to the shopping cart.

The server receives both requests. Bob wants to delete all the items in the shopping cart. Alice wants to add a drill to the shopping cart. Both requests occurred at the exact same time, but we have to decide how to process them in some order. This is a situation known as a conflict. Which request should execute first? Should the resulting shopping cart be empty? Should the shopping cart only have a drill in it?

In either case, Alice or Bob is going to be disappointed. There is no way to avoid that. But we do need some way to resolve the conflict deterministically. We do not want to have to send a message to both Alice and Bob that says, "Sorry, our shopping cart cannot handle your request. Please try again later." We need the shopping cart to be a conflict-free shopping cart.

Today's episode is about the different techniques that can be used for conflict resolution. The shopping cart is a simple example, where a user collaboration leads to conflict. Imagine all the other ways that you collaborate with other users. Chat systems like Slack, social networks like Facebook, document systems like Google Docs, one way to resolve these types of conflicts is through a technique called operational transform.

Operational transform requires that all the operations in the distributed system be funneled through a centralized server. When a conflict occurs, the centralized server detects the problem and figures out how to resolve it.

Google Docs uses operational transform to resolve frequent conflicts that occur when two users are sharing a text document. But operational transform only works if you have a centralized server. An alternative solution is conflict-free replicated data types, which maintain each user's replica of the data in a format that allows the client copies to resolve conflicts in a peer-to-peer fashion without a centralized server.

Here is the last example, and we will get to my interview with Martin Kleppmann eventually, but just this example. Alice and Bob are now collaborating on a document that uses a CRDT data structure under the hood. The document is represented as a conflict-free replicated data type on each of their computers.

Whenever they send their local changes directly to each other, any conflicts that occur can be resolved directly on their client. They don't need a centralized server. Alice and Bob can collaborate on a document, just like they might send e-mails to each other if they both had a e-mail client and e-mail server hosted on their computer.

With CRDTs, we can build decentralized collaborative applications. But CRDTs are hard to use. Just like with Blockchain technology, we do not yet have the simple, elegant abstractions that let inexperienced programmers build peer-to-peer applications without the fear of conflicts.

Martin Kleppmann is today's guest and he is a distributed systems researcher, as well as the author of *Data-Intensive Applications*, which is a fantastic book. Martin is concerned by the amount of centralization in our systems today. He's concerned about the centralization of our computer networks and he works on CRDT technology in order to make it easier for people to build peer-to-peer applications.

Most of the people who know how to build systems with CRDTs are distributed systems PhDs, or database experts, or people working at huge internet companies. How do you make

developer-friendly CRDTs? How do you allow random hackers to build peer-to-peer applications that avoid conflict? Well, you could start by making a CRDT out of the most widely used generalizable data structure in modern application development; the JSON object.

In today's episode, Martin and I talk about CRDTs, conflict resolution and the idea of decentralized application development. This is Martin's second time on the show and his first interview is the most popular episode of Software Engineering Daily ever. You can find a link to that episode in the show notes, or you can find it in the Software Engineering Daily app for iOS and for Android.

In other podcast players, you can only access the most recent 100 episodes of Software Engineering Daily. With these apps, we're building a new way to consume content about software engineer, and you can find all of our back catalog episodes. These apps are open sourced at github.com/softwareengineeringdaily. You can download those apps, there is a link for the Software Engineering Daily apps in the show notes and find all 600 plus of our episodes.

These apps provide recommendations based on the episodes that you have listened to in the past. People have had trouble finding episodes that are appealing to them. Well, these give you a recommendation system based on things that you've already heard.

With that, let's get on with this episode.

[SPONSOR MESSAGE]

[0:10:53.6] JM: You are building a cloud-native application and you need to pick a cloud service provider. Maybe you're just starting out with a new app, but you have dreams of scaling into the next giant unicorn. Maybe your business have been using on-premise servers and you want to start moving some of your infrastructure to a secure cloud provider that you can trust. Maybe you're already in the cloud, but you want to go multi-cloud for added resilience.

IBM Cloud gives you all the tools you need to build cloud-native applications. Use IBM Cloud Container Service to easily manage the deployment of your Docker containers. For serverless applications, use IBM Cloud functions for low cost, event-driven scalability.

If you like to work with a fully managed platform as a service, IBM Cloud Foundry gives you a cloud operating system to control your distributed application. IBM Cloud is built on top of open source tools and it integrates with all of the third party services that you need to build, deploy and manage your application.

To start building with AI, IOT, data and mobile services today, go to softwareengineeringdaily.com/ibm and get started with countless tutorials and SDKs. You can start building apps for free and try numerous cloud services with no time restrictions. Try it out at softwareengineeringdaily.com/ibm.

Thanks again to IBM for being a new sponsor. We really appreciate it.

[INTERVIEW CONTINUED]

[0:12:29.5] JM: Martin Kleppmann is a distributed systems researcher and the author of *Data-Intensive Applications.* Martin, welcome back to Software Engineering Daily.

[0:12:37.1] MK: Hi, Jeff. Thanks for having me.

[0:12:39.1] JM: You are doing some research in the area of distributed systems and I want to start with the motivation for that research. Before this interview, we were talking about the good old days of Microsoft Word, where you would save a file on your computer and it was actually just saved on your computer and it was not shared with anybody, unlike the Google Docs of today, where you save something on Google Docs and it's automatically synced with the cloud.

In many cases, that's a feature. That's very desirable. If our computer gets destroyed, or if we happen to be somewhere without our laptop, we can access our file in the cloud. But under other conditions, this model might be more of a bug. Describe the conditions that you started to think about the research that you're doing today.

[0:13:34.4] MK: Well, so I think about if you create an app – create a file on your local computer with whatever software you're running locally, then you own that data for yourself and you can share it with others if you want. But primarily, it's stored on your own machine.

Whereas now, we're using all of these web apps and services, which are super convenient as you say. But now all of the data, the primary copy is stored somewhere in the cloud, maybe on Google servers and we no longer really have that same sense of ownership or own data. You might be using some service run by a startup that might go bust any day and then you would actually lose access to all of that data, because you don't have a locally runnable copy of the software and you don't have a copy of the data locally.

What I like to get to is a place where we can have exactly the same convenience and into active collaboration that we get with these modern web apps, but at the same use as retaining the ownership of the data. Part of that is also wanting it to work offline, because sometimes you're on a plane and you just don't have an internet connection right now. But partly that, just the sense of ownership as well and the ability to keep the software working, even if the company behind it goes bust.

[0:14:56.8] JM: Let's contrast this with a few decentralization models that people are exploring today. There is a popular episode we did a while ago about a decentralized social network called Scuttlebot.

Apparently some of the most prolific Node.js contributors are people who live off the grid, and they have intermittent network connections and because one of them for example, lives on a solar powered sailboat that goes around the earth and – explore. Likes the idea of living off the grid. This type of person obviously dislikes the centralization of the internet services.

I can tell you, I have my own nightmares about the centralization of internet services. AWS is a single point of failure in our way of life today. If AWS disappeared, what the heck would we do? You talk about too big to fail for a bank. AWS is like – no fault of AWS. They built a wonderful inventive service. It's just that's what happen sometimes.

[0:16:03.0] MK: I find it astonishing. That's like if you want to synchronize a file between your laptop and your mobile phone and they might be 30 centimeters apart from each other. But actually, the best way of doing that seems to be via a data center in Virginia, because AWS just has such a holder for so many apps.

[0:16:21.7] JM: Yeah. We might call that something like a code smell, or infrastructure smell. There's something going on there that seems wrong.

[0:16:28.5] MK: Exactly. Well, people make fun of Github for example, because Github is nice decentralized version control system. Hey, let's put all of our repositories in a centralized service and we'll be able to access it for that.

[0:16:41.9] JM: Right. Okay. Scuttlebot, the way Scuttlebot works is something like people are doing stuff in their social network and the nodes opportunistically share information with each other, whenever you happen to be close enough to another node that is in the Scuttlebot network. You share your timeline updates, or your chat messages and people who are curious about that can go to that episode.

The other decentralization approach is people are exploring these days are around the Blockchain. I'm sorry, having a decentralized transaction management system, and this can apply to Blockchains that are money in the case of Bitcoin or a compute in the case of Ethereum, or file storage in the case of Filecoin/IPFS.

You are concerned with a slightly different challenge of our distributed systems in our data sharing. Give a little bit more color on the difference between the challenges that you're approaching and those of something like a Scuttlebot or a Bitcoin.

[0:17:46.5] MK: Yeah. With the case of Blockchain and Bitcoins, what we have there is as you said, a mechanism for decentralized agreement on transactions, which is a very powerful construct to have. But it's actually not what we need for typical data synchronization.

For example and what I said earlier is I've got some data on my phone, I want to sync it to my laptop. Maybe I have a Bluetooth connection between the two – maybe I'm sitting on a plane

and I'm not actually connected to the internet, but I still want to be able to sync data between those two devices. That's a completely reasonable thing to want to do.

For that, I don't need a Blockchain, because they're connecting to some kind of Blockchain network and getting a transaction registered and waiting for the block to be approved would be complete overkill. All I want to do is sync some data between some devices. I think Blockchains are solving an important problem, but it's a very different one from this kind of data synchronization and offline working that I want to support.

Then with Scuttlebot, I don't know very much about the technical internals of how it works, but it sounds to me like it's mostly about one person can post a message or update of some sort and that gets synced to other people then who follow that person. That is great. That kind of data synchronization works really well, because there's a single author of the message of the update. Then the system just needs to figure out who needs to receive a copy of that message.

However, if you think about the Google Docs for example, or spreadsheet, in that case you've actually got this shared data structure that several people can edit at the same time. That's a bit different then from just posting an update, because the update has a single author. Whereas, a Google Doc may have multiple authors and they might be changing things independently of each other in different places on different devices. There, we still need to make sure that we end up with a consistent document at the end.

[0:19:54.1] JM: The area of computer science that we could label this problem might be conflict resolution. Two people are collaborating on a document, or collaborating on anything, like a document. I've got hello world. You delete world – you delete the word world, while I am adding my name as Jeff to the document. Should the document then say, "Hello world. My name is Jeff," or should it say, "Hello. My name is Jeff."

You could make arguments in either direction and we'll certainly get into that. I want to work our way slowly towards the discussion of resolving that kind of conflict; the conflict in a distributed document, but maybe some easier examples for people who are less familiar with distributed systems to understand. Why don't you give a simple example of a conflict that can emerge in a distributed system and the way that we might resolve it?

[0:20:51.0] MK: A classic example that's often used is Amazon shopping cart. The case there was that you have maybe different computers making updates to a cart, like adding stuff to a cart, or removing things, or changing the quantity of items. Potentially, those updates can go to different servers, different database servers. So you can end up with actually different servers having different versions of the cart.

On one, the product A has been added, on a different one, the B has been added. Now after the connection is restored between those, you want to get everyone into the same state and probably that would be a case where both A and B have been added.

This is an interesting example that's often cited, because you would think that the easiest way of solving this is just to take the union of whatever you have on the different copies. That is if one server has A in the cart, another server has B in the cart, then the result is that you should have A and B when you merge them together.

However, the tricky bit with this is if you also want to support removing things from the cart, and so it could be then that actually the starting point was that you had A in the cart, and then you came along and wanted to remove A and instead add B. Now, if you just do this union to merge together the different copies, then actually make the A item reappear in the cart, even though you actually wanted to delete it.

That's where this data structure start becoming more difficult. If you want to support things like deletion, or if the data structures are more complicated, then just like a set items, but you want to maintain the order, or you want to do text editing, you want to have some kind of tree-like data structures, all those things start becoming trickier.

[SPONSOR MESSAGE]

[0:22:55.8] JM: Today's sponsor is Datadog, a monitoring and analytics platform for cloud-scale infrastructure and applications. Datadog integrates seamlessly with more than 200 technologies, so you can track every layer of your complex microservice architecture all in one place.

Distributed tracing and APM provide end-to-end visibility into requests wherever they go across hosts, containers and service boundaries. With rich dashboards, algorithmic alerts and collaboration tools, Datadog provides your team with the tools that they need to quickly troubleshoot and optimize modern applications.

See for yourself. Start a 14-day free trial today and Datadog will send you a free t-shirt. Go to softwareengineeringdaily.com/datadog and get your free soft t-shirt.

Thanks for listening and thanks to Datadog for being a sponsor. Let's get back to the show.

[INTERVIEW CONTINUED]

[0:24:01.7] JM: At a fundamental level, if two transactions occur at the same time and then they both hit the server and the server decides on some ordering of those transactions, the ordering of those transaction is going to lead to what the end result of that conflict resolution becomes. Does that just lead to a situation where the server is essentially making a subjective decision and we have to put rules in place around how a conflict is resolved? Or can we reach some objective truth about how a conflict should be resolved?

[0:24:41.7] MK: Well, to the classic way of doing this in databases is with serializable transactions. In that case, you do really have the database putting an ordering on these things. For example, you may know that first A was added to the cart, then A was removed from the cart, then B was added to the cart.

It's very clear that adding A happened first, then removing A happened second. Definitely the end result should be that A is removed, and so it doesn't then start reappearing again. Serializable transactions are great if you can afford them. But once people start to scale these systems into larger and larger things potentially with geographic replication, you run into the problem that you now have to send all of the updates to a central server, or you need some kind of leader or master database, which actually decides on that ordering.

That unfortunately runs counter to wanting very high availability. If you want to be able to have the system continue working even when bits of the network are disconnected from each other,

you have a network partition, then you actually want different servers to be able to independently accept rights.

Now in this case, you've got two different servers and the ordering of the rights that happen on those two servers is no longer clear. What we're dealing with here now is a divergence between these two. We can define what constitutes a valid merge once the two come back together again. Sometimes, it's not entirely obvious like what a correct merge would look like.

[0:26:27.6] JM: This has been a problem in computer science since the 80s, since I think the late 80s when people started talking about this seriously, maybe even earlier. Is this a solved problem, or what's – give a history of the evolution of attempt to solve this conflict resolution problem set.

[0:26:47.2] MK: There's a surprising amount of drama actually. You wouldn't feel it, but people have been looking at this for a long time. I mean, the part of this is like the CAP theorem way of thinking, which actually goes back to the 70s as well, where people realize like, "Oh, we have copies of these data on different nodes. They might both get updated, now we have conflict that we need to resolve." The initial version started out just very simple with maybe version numbers and getting the user to write their own code to merge versions together. But like we saw in the Amazon shopping cart example, it's not always actually that simple to do that merge.

Now there's one liner of research in this area called operational transformation, which does go back to the 80s, where people studied mostly the problem of collaborative text editing. That's what you get with Google Docs, where you've got several people contributing to some text document.

What you want to do there is that each user has a local copy of the document on their own computer. Nowadays, it's in our web browser. Whenever you type a letter, you want to make that change immediately to your local copy. You don't want to have to wait for a network roundtrip before the letter you type appears on the screen.

You apply the changes locally immediately, but that means then asynchronously some later point that change gets applied to other people's copies of the document on other computers.

Now during that little period of time after you apply a change locally and before it gets sent over the network, you can get divergence, so the document can drift apart slightly. Different users, use of the document can drift apart slightly.

It's even more extreme if you want to support offline editing. If you allow people to change the document offline then they might have a whole bunch of changes made on one computer, and a whole bunch of changes made independently on a different computer without knowing about each other. Then at some later point, they both come back online and they resynchronize.

I said operational transformation has been studied for a long time and there were a whole bunch of algorithms that have been proposed precisely for this problem of text editing. The problem with most of these algorithms is that they later turn out to be wrong. It's actually turned out to be incredibly subtle problem, even just a simple problem of text editing, where the only thing we allow is inserting a letter at some place and the text – or deleting a letter. Those are the only two operations.

Nevertheless, some researchers would come along, propose an algorithm, and then two years later some other researchers realize, "Oh, there are actually some cases in which these algorithms fail to converge." There's simply some cases in which you can construct an order of operation such that at the end, they don't have the same document. It doesn't become consistent. It remains permanently inconsistent.

Researchers purported different algorithms and solve this bug. Then two years later, some other people find yet another problem. Again, there's some circumstances in which they fail to be dodged. If you through it the literature on operational transformation there are like I think five failed algorithms that have all been published in good academic venues. They've all had peer review and nevertheless they're simply wrong.

There are a couple of operational transformation algorithms have survived this disaster. Google Docs is now actually using one of the remaining ones that actually turned up to be correct. It does so at the cost of send all of the changes via a central server.

Some of the operational transformation algorithms tried to support these kind of decentralized architectures, where you could do peer-to-peer synchronization of data. Almost all of the peer-to-peer operational transformation algorithms just went wrong.

The problem becomes a whole lot easier if you assume a central server, and that's what Google has done with Google docs. That's okay. It works, but it does mean that you can't have two people editing while they're disconnected from the internet and just synchronizing via a network. That won't work, because really the fundamental assumption of the algorithm using Google Docs is that you can send everything via the central server that is hosted by Google.

Then people started investigating a new family of algorithms called CRDTs, that stands for Conflict-free Replicated Data Types. That was really like an allergic reaction to operational transformation basically saying, "The entire history of operational transformation research has just been such a train wreck. We're just going to start afresh with a completely different model."

People have proposed a number of CRDTs for text editing, but then also for other kinds of data structures, like maps and sets and JSON. That has been a whole lot more successful. There hasn't been this kind of catastrophic failure of algorithms with CRDTs as there was previously with operational transformation.

However, with CRDTs the challenge has been more on the performance side. That some of these algorithms are too slow, or require too much metadata overhead. They're not practical yet. This is actually the area in which I've been working in particular trying to understand CRDTs better. Check that they really do work correctly and improve their efficiency, so that we can actually start building applications based on them.

[0:32:34.6] JM: Just to give people some gravity for this problem set, the shared document problem is almost a base case of problems. Because if you can imagine, if we want this ability to have decentralized file collaboration, we would not only want to collaborate on say a Google Doc, where it's not – or I shouldn't even say Google Doc, because obviously this would not be a document.

We would not only want to collaborate on a document where, okay if somebody – if we have a conflict and a conflict resolution uses an operational transform algorithm that results in a misspelled name, or a typo, or some grammatical error, it might seem like, "Okay, it's not a big deal. Like okay, this works 99 times out of a 100. We just end up with a typo sometimes. It's not a big sacrifice to make. Let's just use the operational transform algorithm and get our decentralized world and that's fantastic.

If you started to collaborate on something like a blueprint for a building and you wanted to have a decentralized way of collaborating on blueprints, if that kind of thing gets a typo, that could result in like the building falls down. That's a serious issue. I just say that to motivate our discussion a little bit further and we'll get into the CRDTs in a second.

As far as the operational transform and really this whole field, I've done some shows about the Paxos algorithm, which is really about reaching consensus in a distributed system. I actually had the privilege to interview Leslie Lamport several years ago and it was incredible. Why isn't this a subset of the problem of consensus? This seems like a distributed consensus algorithm. We just need to come to a consensus on what the reality is. I thought Paxos did that.

[0:34:31.2] MK: Yes. It does seem like this would be very similar, but actually they are different at a very fundamental level. The type of consensus that you get from something like Paxos is as follows; you can have one or more nodes propose a value, and then those values gets decided. The expected outcome of the consensus is that everyone decides on the same value.

This applies by the way to Blockchains as well, so that consensus is about what is the next block going to be. In particular, what transactions appear in that block? The consensus there is about did a particular transaction happen or not?

This is actually not what you wanted in the case of document editing, because if you apply consensus to document editing it would be like, I have a bunch of changes to document. You have a bunch of changes to the document. One of the two of us is going to get their changes selected and the other one is going to throw it away, which is what he wants in case of transactions. He want money that can only be spent once. You don't want conflict of money get spent twice.

In the case of document editing, consensus is really not what you want, because it would mean that only one of our changes get through and the others will get thrown away. What we want to do is we want to get into the same state, we want to reach a consistent state, but we want states to actually reflect all of the changes that have happened and we want those to be merged together in some sensible way. That is what CRDT has given us.

[0:36:11.8] JM: Right. This is like, I branch – I'm a Git branch and I need to eventually merge that back into master. The result of that merger is going to be a completely new thing. We are coming to a consensus, but it's a consensus that is the unification of our two disparate views on reality, rather than the Paxos model which comes to a consensus on just one of the two perhaps disjoint sets of reality.

[0:36:43.9] MK: Yeah, that's right. Actually Git's branches and mergers is a really good way of thinking about this. A lot of the history tracking we do in CRDT is actually looks very similar to what happens in Git. The main thing we're trying to automate there is this merge process, and that is what is like to resolve a merged conflict if two people have edited the same code. Gets pretty tricky.

Most users in most cases won't want to have to deal with something that looks like a Git merge conflict. It's much easier to just merge the changes together automatically. In most cases, people are editing different parts of the document, and so that's perfectly fine.

The tricky cases only happen if people are like literally editing the same words in the same sentence. In that case, well actually what Google Docs does is it just picks some resolution in which the insertion and the deletions have happened.

The result might not actually be a grammatically correct English sentence and it doesn't even try to be clever about English grammar. The best it can do there is just ensure everyone ends up in the same state, and the spellchecker will then tell you if some of the merged conflict has produced some weird words that they're not actually English.

That is the level we're working at as well. We can't fundamentally understand the English language structure of the text. Trying to do so would just leap down a very complicated natural language processing problem, so we'd rather not even try to solve that. But just get everybody into the same state and maybe highlight if people made changes very close to each other just to then double check the merged result is actually sensible and meaningful.

[0:38:35.6] JM: We've laid the foundation for a conversation about CRDTs. Multiple users are changing a replicated document. Those changes can result in conflict, whether it's talking about conflicts in a Git situation where we need to merge those, or conflicts in a text document where you have made changes to the document that conflict with the changes to my document.

We should try to represent this document in a format where conflict can be resolved. We use Conflict-Free Replicated Data Types. What is a Conflict-Free Replicated Data Type?

[0:39:11.5] MK: Essentially, it's a way of giving you a data structure that you can work with and that you can modify in certain ways. Such that when several people change that data structure concurrently, the results can be merged automatically.

Like one data structure that you can work with if it's a text document is a list. A text document we would represent as a list of individual characters. You can modify this text document by inserting characters in certain places, or deleting characters. That is enough to represent plain text already.

If you have a more complex type document, say a spreadsheet or like the architectural drawing of a building like you were saying earlier, that would then be some larger data structure. So in the case of a CAD program for drawing, I think that would probably be some kind of tree of objects that's typically the way vectographics works that you have some kind of tree of graphical objects that can be grouped together.

Related things are grouped together into a sub tree, then you could potentially have people working on different parts of these tree, like one person is adjusting the shape of the window while another person is working on the water pipes that come into the building. Those two things

can probably be edited independently from each other and can be merged together without too much trouble.

Whereas, Git works purely on files that it doesn't interpret it. Just treats a file as a sequence of bytes. For merging, it treats it as a series of lines and you can just do line level mergers. With CRDTs, we thrived it to more application meaningful data structures. Some of the work we've done is on JSON, so a lot of data can be represented as JSON essentially.

JSON is actually fairly simple format. You can imagine it as a tree. As you know, the two construct of JSON does the curly braces, which gives you a map, like a key value mapping and there's the square brackets, which gives you a list of constructs. You can nest those two things inside each other arbitrarily, so you can have a list of maps or a map where the values, or list, or various combinations of those things.

By expressing data at the level of a data structure, then we can define sensible merging rules. For example, if we both edit a map, if you insert key A with value A into a map and I insert key B with value B into a map, then we can merge those two nicely. In fact, this looks very much like the shopping cart we talked about earlier; with a list, if you insert something, I insert something, we can merge the two, so that's both of our insertions are preserved and so on.

[0:42:15.7] JM: What are some of the benefits of the CRDT approach relative to the operational transform approach?

[0:42:22.0] MK: The main one really is that the algorithms work without assuming the central server. As I said previously, people have tried to make operational transformation algorithms that work in a peer-to-peer setting without going by a central server, but my system failed.

CRDTs are really a fresh start that allowed us data synchronization without assuming anything about the network topology. Without assuming to kill a server, you can go serverless, literally serverless in the sense of actually not having any servers, because you can synchronize data via a local Bluetooth connection, or via a local Wi-Fi. It doesn't have to necessarily go through some kind of central node.

[SPONSOR MESSAGE]

[0:43:16.9] JM: G2i is a talent platform built for engineers, by engineers. React Native, React and mobile, the developers on G2i have expertise in the best tools to build your applications. When I need engineers to help me out with my apps, G2i is the first place I go, especially when I'm building with React or React Native.

Contract a G2i developer to help you on a short-term basis, or hire a G2i developer fulltime. If you're looking to build cross-platform applications in React Native, definitely check out G2i. The G2i platform is a community of React Native, React and mobile developers. These engineers can become part of your team.

If you're looking for developers to build your product, check out g2i.co. That's the letter G, the number 2, i.co. You can also send me an e-mail and I'll be happy to tell you more about my experience with G2i. Find your React Native, React and mobile talent by going to g2i.co.

Thanks to G2i for helping me ship my products and thanks for becoming a sponsor of Software Engineering Daily.

[INTERVIEW]

[0:44:40.6] JM: What's the sacrifice there? In contrast with the Google Doc's approach, where Google centralizes all the changes that funnels them through a central server, do they end up with your typos that way, or – I assume there is some matter of correct, or implied correctness that they get out of that centralized server approach.

[0:45:07.4] MK: In terms of the conflict resolution, I believe what Google Docs does is pretty much the same as what the CRDTs do. In terms of the consistency of the outcome, like if two people make – correct the same spelling mistake at the same, for example you will get the same kind of weird outcome in both Google Docs and CRDTs.

One example does I like testing is what happens if one user deletes an entire paragraph and another user simultaneously changes one word inside that paragraph? I think what Google

Docs does in this case is actually the end result is a document where that paragraph is missing, except for that one word that was added to the middle of the paragraph is still there, even though that the surrounding paragraph was deleted.

This is a bit weird, but a lot of people seem to use Google Docs and manage just fine. My thinking does CRDTs will manage just fine as well. The tradeoff there mainly is actually around efficiency. That's with the CRDTs that at least the first generation CRDTs that were developed for this, they require a lot of extra metadata, so the document where you have every single character is an editable item, a character you can represent in one byte. But you might then suddenly need 20 bytes of extra overhead of metadata attached to every character, and now suddenly your document has grown 20X in size.

That is really the tradeoff there. That's the operational transformation algorithms have been a lot more efficient. What I'm working on right now in my research is actually is how far can we push down that metadata overhead with CRDTs that we can make them really efficient enough to be of practical use without that 20X overhead. In my latest experiment, I actually got it down to about 1.7 bytes overhead per character in a text document.

[0:47:12.4] JM: Well, that sounds great.

[0:47:13.1] MK: That is doing a lot better than the 20X or 100X we were seeing previously.

[0:47:19.1] JM: Tell me if I have the analogy right. With Github, you get basically the commit history of your entire project. I'm sorry. I think I said Github. With Git, you get the entire history of your project, and at any given time I can roll back to a certain situation in that document history. You're talking about creating basically a document model for basically any document that if I understand correctly, the metadata, what you're calling the metadata is essentially is like a commit history, so that any disparate set of documents have enough historical data that they can resolve the conflicts that they might have. Is that a correct analogy?

[0:48:06.6] MK: Yes. It's very much a Git repository actually. I've been working on a implementation that we call automerge, which is a Javascript implementation of CRDTs. It works very much like that. Actually, it does in fact keep the entire edit history. Every single letter that

was added or removed to a document, it stores that and it just puts it in a compact representation, so that you can actually look at the document past moments in time and do this kind of time travel through different versions of the document and see who made what edit at what time.

[0:48:43.2] JM: How do you decide when to throw away old versions of a document? Do you just keep the entire thing? Or what's your approach?

[0:48:51.4] MK: At the moment, we're just keeping the entire thing, which is by the way what Git does as well. With the Git repository if you clone it, unless you do a shallow clone, you actually get the full commit history of all changes ever made. That actually seems to work surprisingly well even for large project.

Right now I'm working on the basis that we'll just try to keep all changes, as long as we can and just represent that in a compact way, so that it doesn't cost us too much extra storage to keep all of that history.

That's really just one case where you'd want to deliberately throw away history. That is if I send the documents to somebody else and I don't want that new person, that new collaborator to see the entire editing history. Because maybe there was some embarrassing stuff that I have in a past version of the document that was then deleted. So in that case, I don't want to actually share the full editing history with someone. I just want to give them a snapshot of the latest version.

For that at the moment, what we can just do is copy it all into a fresh document and start afresh. There is probably some better things we can do around creating a snapshot that gives you the current state without the full editing history, but which still remains compatible, so that when somebody then edits the snapshot version, you can still combine those edits with the version that has the full history.

There's similar research to be done at the moment there. For now, except for this previously case, I think it's actually okay to just keep the full history. Suddenly for stuff that is edited by

humans, there's only so much that a human can type. Computers can store a lot more data than what we can type.

[0:50:40.8] JM: The Blockchains that I've explored on the show often use a Merkel tree to do – it's like a form of compression of compressing a history of changes that have occurred. Do you use a Merkel tree type of structure for the CRDTs?

[0:51:01.1] MK: Not at the moment. There are some implementations that use some of these Merkel tree-like ideas. At the moment for our purposes in auto-merged the type of document we're talking about is something like a Word document, or a spreadsheet, or something like that, which is fair more – the total amount of data is small enough that you can fit in memory on one computer.

For that, you don't really need these Merkel tree-like structures, because if you want to give somebody a copy of the document, you just give them a copy of the entire thing and that's fine. It's like deliberately small data and not big data.

You could definitely imagine one thing to generalize this to much larger data structures, like I imagine say, "Could we build something like a Facebook that is actually built upon CRDTs?" In that case, you certainly wouldn't be able to download the entire social network data to every person, so then you would have to download subsets of it. Then those cases, then techniques like Merkel trees would become relevant.

[0:52:08.4] JM: Could we go through a simple example of a conflict that could occur between two instances of a document and how that conflict is resolved at the data structure level, how that data structure would be represented and what would go on in the resolution algorithm?

[0:52:27.9] MK: Sure. This would be easier with a white board.

[0:52:29.3] JM: I know it's hard to describe over voice. I know, I know. I will certainly put things in the show notes that people can explore. You don't need to go into gratuitous detail that would be impossible for people to visualize.

[0:52:42.9] MK: Yeah. I have some slides, which give some examples of these which we can put into show notes. I'll describe it in words. One interesting case is you have a text document and you have two people who are both adding text at the end of the document. Right at the end of the document, you put the cursor there and I type one paragraph and you type another paragraph.

What we expect at least is the final merged result has both of our paragraphs, and it's either your paragraph first than mine, or mine first than yours. It doesn't really matter in which order they occur, because there's nothing really that determines what the right ordering should be, but we expect them both to be there.

This is actually an interesting case, because some of the CRDTs don't handle this particular situation very well. Some of the CRDTs I've looked at will actually take the individual letters from both of our paragraphs and interleave them potentially. You would end up with a jumbled mixture of letters; some of the letters coming from my paragraph, some from your yours, but the end result would not be readable text. It would be some kind of mad jumble of the individual letters, which is not really a good outcome in this conflict resolution.

The algorithm that we've been working with doesn't have this problem. That algorithm, we have put either one paragraph first and then the other one after that, but it doesn't intermingle these three. The way this works is you can imagine that each letter in the document a unique identifier. That identifier consists of a number and a node ID. The node ID is like some kind of identifier that is the copy of the document in my browser, and in your browser you would have a different identifier. This reserves to make them unique.

We can now create unique IDs for each character by incrementing the counter, the number part. Every time I type a letter I just give it a number that's one greater than what we have previously, and we'll attach my node ID to that. You can do the same. We might end up generating identifiers that have the same number, but they would differ in the node ID part, and so they're still unique overall.

Now what we can do is we can actually use those numbers and node IDs to determine the final ordering of things in the document. The exact rules for how you do this is a bit subtle, but the

basic idea is that in order to decide whether your paragraph comes first, or my paragraph comes first, we look at those numbers and we look at the node IDs.

We first look at the number. If one number is greater than the other, we put the one with the greater number first, the one with the lesser number second, so we order in descending order of those numbers essentially. If we end up with the same number, then we actually compare the node ID part. Again, we just put them in descending order.

That guarantees a deterministic outcome. No matter in which order you apply these different editing operations, at the end everyone is going to order the insertions based on those unique identifiers and everyone sees an ordering of those unique identifiers. That is what guarantees that everyone actually ends up in the same state at the end.

[0:56:23.8] JM: One of the examples that I heard you give in one of your talks I saw you give, and again for people who are confused, it's definitely a great source of information is the talk that I'll put in the show notes, or the multiple talks that I'll put in the show notes. You used these examples to motivate the fact that there are not clear APIs for concurrently editable data that are not confusing.

My sense of talking to you about this is – that this is actually important, because there is an element of subjectivity to how we're going to resolve a synchronous change, a change that occurs at the exact same time. The way that we resolve that is going to depend on the application. Like we want to do different things. Maybe if conflicting changes occur when we're concurrently editing a blueprint for a building, maybe really want to just signal the user, "Hey, there have been conflicting changes." You want to do something about this. You want to go talk to that person psychically or go give them a phone call to resolve this. You maybe don't want just the automatic reconciliation algorithm. Is that correct?

[0:57:37.0] MK: Yeah, potentially. There's certainly some cases where you might just want to bump it up to the user and ask the user to solve it. It's also true that it's not always obvious what – if you do want to merge automatically what is actually the right way of doing this merge. What I like to get to is that we can actually have the CRDTs just available as a library that anyone can

build applications above, and you shouldn't have to have a PhD in distributor systems to make sense of those.

We really need to figure out like a programmer-friendly, developer-friendly way of expressing these data structures. That is tricky, because fundamentally if different people can change stuff at the same time, sometimes really weird things happen. We need to translate these really weird things that can happen somehow into a set of APIs that will actually make sense to people.

I think we're gradually getting better there. With this auto-merge Javascript implementation that I described, I actually did a collaboration with a few folks who were building an actual example app on top of it.

That was great for learning about what assumptions are the app developer is making about how this APIs work, what kind of things that I get confused about, how can we make it better, how can we just design the API in such a way that it seems obvious and nobody gets too confused by it, even though what's going on underneath might actually be quite sophisticated. For that, it's been really useful actually collaborating with people who are not CRDT experts, but just want to build an app and who don't care about the details of what happens internally.

[0:59:22.6] JM: With other data structures like a map, for example, we've developed a vocabulary for doing things with a map; put and get and delete operations. Are you starting to get a sense for what the vocabulary for a CRDT is going to be?

[0:59:40.2] MK: Yes. Ideally, I would like it to look exactly like your familiar data structures, like as you say with a hash map you can put, you can get, you can delete. That's nice. Let's take an example of a case that I rangled with, which I'm still not entirely sure actually what the right result looks like.

You can use map to represent, say an item in a to-do list. A to-do list is like a list of items, and each item will have a title that's like a text of boil milk, or water the plant and then maybe a bouillon, which indicates whether it's been checked off the list or not, maybe have it timestamped or a deadline or various other stuff attached to it.

We initially designed a JSON, a CT algorithm. It had this weird property that if one user deletes an item from our to-do list and another user at the same time updates the check mark, the bouillon flag of whether it's done or not, then the merged outcome would be that you have this item in a to-do list which consists only of the bouillon field that where the title has disappeared.

This is really weird, because nobody really expect an item in a to-do list without the text of what that item is. In that case, really what we probably want is that the deletion takes precedence. If the one item was deleted from the to-do list and somebody simultaneously updated that item, we're just going to forget about that update, because the item was deleted, so we don't care about the fact that it was changed anymore. It's just gone by that point, unless somebody doesn't undo. That's one example of where we ran into an issue.

Now it seems like a reasonable way of handling, this is to just let the deletion take precedence. But another case happens where you have, say different people creating a map at the same time. Let's take as an example, say you wanted to implement Slack. You own version of Slack and a message in Slack is again an object and it has the text of the message and it has a field indicating who wrote the message and maybe have the timestamp.

Then Slack added this ability to add emoji reactions to messages. I don't know if you're familiar with that. You can have like a –

[1:02:09.7] JM: Yeah. I use that on an hourly basis.

[1:02:11.4] MK: Great. This makes a nice distributed systems example, emoji. You have five people who added the smile emoji reaction to a message and three people who added the celebration emoji and a couple of others, whatever.

We can again represent that as a map. Let's have, the top level we have a message object which has a key, which is text, a key which is timestamp, a key which is author and a key which is reactions. Under reactions, we have a nested map. That map is smiley :5, celebration :3, star :15, whatever. That way, we're packing all of the reactions together into one object.

Now that is okay, but imagine what happens if you go back in time to the day before Slack actually have this feature of the emoji reactions. In that case, that reactions field doesn't exist. Now the first time somebody make other reaction to a message, it first got to initialize that reaction's field and puts an empty object, or empty map there, then fill that map with the first reaction.

Now what happens if two people simultaneously decide to add the first reaction to a message, so now you've got two people independently assigning a new empty object to this reaction's field, what do you expect to happen in that case? Do you want to merge those two objects together, or do you just want to say, "Well, these are two independently created maps. We are just going to keep them separate." One of the two is going to win, the other one is going to be overwritten.

That's really tricky, because if you start going down the case where, okay we're going to merge these maps together then you end up going towards this problem that I entered earlier with a to-do list item that has a bouillon, but not the title.

If we say we're not going to merge these two maps together, which is going to keep them separate, then in some cases, you're going to actually lose data in this case if two people create that first emoji reaction simultaneously, only one of them is actually going to be preserved in the final data.

That is the kind of irritating things that we've been grappling with. I don't know, maybe I'm overthinking it. For now, we just got like a simple solution that seems to work well enough. I guess, we'll have to just try – people will have to start building apps on top of these types of data [inaudible 1:04:43.1] and we'll just see what sort of issues and bugs people run into.

It's exactly, you run into all these kind of interesting scenarios that just don't happen when you're just writing code on a single machine, like this whole issue of different people concurrently creating the first reaction. Like in a single-threaded case, that just doesn't happen. We're used to thinking very sequentially about the way we write our software. Suddenly, if you're allowing this collaborative editing, you're moving into the space where concurrent changes happen and concurrencies – it's been hard for people to reason about.

[1:05:23.4] JM: All right. Well, you've been very generous with your time. I want to wrap-up with just a question of bringing this to market. You are doing research on this and you're a public research figure, so I'm sure the words that you say eventually make their – actually, they probably make their way in the industry very quickly on a natural basis.

Do you have any vision for how – this JSON CRDT. The world is moving towards Javascript obviously, Javascript is eating the world and JSON is eating the world in terms of data representation. Do you have a vision for how your conflict-free replication JSON data type might make it to market and start to make it into an application like Slack?

[1:06:06.5] MK: Yeah, totally. As I said, we're working on this Javascript implementation called Automerge, which is it's by no means production ready yet, but it does work and it's getting gradually more efficient.

I spent several rounds of iterating under-performance of it. I think so far speeded it up about three orders of magnitude from the initial version, from which [inaudible 1:06:31.4] use that the initially version was extremely slow. Now it's starting to get fast enough that you can actually build some reasonable applications on top of it.

It's research code and we're using it for writing papers and doing performance measurements as well. But I am hoping that this will become good enough that people can actually use it in production for real to build real applications. That is the trajectory. We're heading down there.

I believe this wield a lot. We need not just the CRDTs, but another layer to it is figuring out the networking as well. We've got a prototype that uses web RTC for peer-to-peer communication between different devices, which is really neat. We can actually do surprisingly much without a server or using just a web RTC signaling. But beyond that it's just doing peer-to-peer synchronization.

The downside of that is that you need people to be online at the same time in order – it's just like a video call or something like that. It only works when people are both online and can exchange edits at the same time.

I think we'll probably still want cloud services, which will then act as a buffer so that I can upload my changes to the cloud and sometime later when you come online, you can download your changes again. That can actually be end-to-end encrypted, for example. We don't actually need the servers to be able to read the data that's being exchanged there.

It's really just using the server as a buffer of messages. I think of this is the kind of cloud-optional programming model, where it's nice to use cloud services for exactly this kind of storing of data. But we also want to be able to use local works for synchronizing data when available. If I'm sitting on a plane, I want to be able to sync data between my phone and my laptop, even if both of them are actually not connected to the internet, because I'm on a plane right now.

That is the world I want to get to, where we can build applications, where you can just freely synchronize data between devices using whatever networking medium happens to be available right now. Where everything continues working offline, like seriously sometimes offline is just really good, where we own it ourselves. So where we have a copy of the data on our local devices, where we still have commercial apps, because obviously developers have some business model somewhere.

If the developer goes bust, I want the software to still continue working. I wanted not to have to rely on the running of some service that I don't control. I don't want to have to deploy my own service, because I don't want to act as admin as well. I just want stuff to be able to synchronize between my devices using the software that's running on my devices, which is an old fashioned way of thinking about software. But I think we'll probably come back to liking that kind of way.

That's really my vision there, that we have this control over data that the apps work offline, they can synchronize in whatever way we like. We have some kind of programming model for building these apps, which is simple. Because that's really I think the stumbling block at the moment.

Right now you actually need a PhD in distributed systems to make this stuff work and we need to get to a point where it's just as easy to build these decentralized apps as it is to throw to get

that rails web app, for example. Like building a centralized rails web app is really simple, there are load of drills, there is good libraries, there is good tool support around that.

What I'd like to get to is that we can do the same for these decentralized apps that is just as simple to throw together a simple and it works nicely. Hopefully that's a future we can reach. It might be still another year or two, maybe three away, but I think we're heading in that direction.

[1:10:35.4] JM: That is exciting talking to people like you and also people in the Blockchain community who fully realize that the APIs are not yet there for building the internet of money, but they are so determined to make it work. People are so determined to rid themselves of the burdens of centralization.

You see the failed efforts of governments for example, to rein in that decentralization. It's fascinating to watch those failed attempts. I imagine that industry is going to be a little bit smarter and they're going to realize well, actually people want this and maybe at a certain point there will be some inflection and industry will start to actually think about, "Okay, how can we productize this? Or how can we give people what they want?" That could be a really interesting point of inflection, maybe for another show.

[1:11:26.0] MK: Yeah. Right. I spent a while, for example talking to journalists. Journalists sometimes work with really sensitive data. If there is a whistleblower who's come to them, source for some story. Journalists are very cautious about wanting to protect the identity, because they might run into trouble otherwise.

That is actually a nice example of where you want the convenience of something like Google Docs, because journalists are working together on some article and just need to get the article ready. Then to annotate the source materials, maintain some kind of knowledge base around that stuff. But actually it's sensitive data and you don't want to just blankly store it in some cloud service.

That's an example of where we see these kinds of tools being actually really important. Not just a nice to have, but actually important for people's security that we can offer a better security

model than what the web apps do today. That's what I'm hoping will be another benefit for of this decentralization as well.

[1:12:27.1] JM: All right, Martin, well, it's been great having you on once again. I can't remember if I mentioned to you, but your previous episode is actually the most popular episode of Software Engineering Daily.

[1:12:36.6] MK: Seriously? Wow.

[1:12:38.5] JM: Yeah. It's funny. I don't know if this is a Zipfian Distribution, but it's almost 2X as popular as the second most popular episode. It's like you are a beloved speaker and I really appreciate you taking the time to come on the show once again. You did not disappoint. I really enjoyed talking to you. It's always very educational. Thank you so much.

[1:12:59.8] MK: Well, yeah. Thank you to all the listeners. I am always very pleased if the random rambling ideas that I talk about are actually helpful and useful to people. Hopefully it will be again with this one.

[END OF INTERVIEW]

[1:13:14.2] JM: Are you a Java developer, a full stack engineer, a product manager or a data analyst? If so, maybe you'd be a good fit at TransferWise. TransferWise makes it cheaper and easier to send money to other countries. It's a simple mission, but since it's about saving people their hard-earned money, it's important.

TransferWise is looking for engineers to join their team. Check out transferwise.com/jobs to see their openings. We've reported on TransferWise in past episodes and I love the company, because they make international payments more efficient.

Last year, TransferWise's VP of Engineering Harsh Sinha came on Software Engineering Daily to discuss how TransferWise works. It was a fascinating discussion. Every month, customers send about 1 billion dollars in 45 currencies to 64 countries on TransferWise. Along the way,

there are many engineering challenges. So there's plenty of opportunities for engineers to make their mark.

TransferWise is built by self-sufficient autonomous teams. Each team picks the problems that they want to solve. There's no micromanagement, no one telling you what to do. You can find an autonomous, challenging, rewarding job by going to transferwise.com/jobs.

TransferWise has several open roles in engineering and has offices in London, New York, Tampa, Tallinn, Cherkasy, Budapest and Singapore among other places. Find out more at transferwise.com/jobs.

Thanks to TransferWise for being a new sponsor of Software Engineering Daily. You can check it out by going to transferwise.com/jobs.

[END]