ActivityPub - one protocol to rule them all?

2018-02-01 activitypub, diaspora, opensource, software, standards

AcivityPub, a protocol that aims to unify federation and client interactions within decentralized social networks, reached its W3C Recommendation state recently. Along with their state change, they got a bit of press coverage, and obviously, the specification is known within my bubble.

Naturally, as I am one of the people behind diaspora*, I get asked what I think about ActivityPub and whether diaspora* will change its implementation to be compatible with it. I could be cheap and just link to xkcd-927 and be done with it, but that would neither be useful to anyone nor respectful towards all the people who worked on ActivityPub, so here we are.

Preface

Distributed social networking is hard. Not only are we facing a lot of very tricky challenges whenever we implement a feature, we also have to deal with a lot of different beliefs and perspectives. In itself, social networking is a very deep subject with a lot of different approaches. Just compare Facebook and Twitter! Both can be considered social networks, but their approach could not be more different. One is designed to focus on person-to-person communication and sharing items with a “friends only” group, while the other is designed to have a public, more open, communication approach. It is very hard, if not impossible, to find a “perfect” approach to social networks. In my opinion, all different approaches have their place and their use, so I usually do not discuss this.

When talking about distributed social networking, you not only have to figure out how your network should look and feel like, you also need to work on the deep, technical issues. How to get content distributed. How to ensure privacy. How to ensure security.

How to maintain good usability, even when the technical aspects are way more complex than they would be in centralized system. So, different approaches to this issue came up with their own solutions. As for diaspora*, we came up with a protocol based on some already existing specifications (like, just to name one example, WebFinger for profile discovery), with some custom extensions to fit our use case. Other networks did their own work on that.

ActivityPub is a protocol built by the W3C Social Working Group[1] that aims to unify communication in federation social networks, with the ultimate goal to have *all* distributed social networks exchanging content with each others. ActivityPub itself outlines some generics, mainly how objects should be exchanged, but it by itself is not the full story. ActivityPub is based on Activity Streams 2.0, which is a specification defining the representation of actions and objects based on a JSON format, as well as the Activity Vocabulary, which defines the actors, actions, and objects that can be used within ActivityStreams, and thus, within ActivityPub.

ActivityStreams is not a new invention. In fact, ActivityStreams 1.0 is a published specification since 2011. It got some implementors, but since it was only used as a presentation format for data, it never took off. ActivityStreams 2 introduced some changes to that, while keeping the main idea.

Generic idea

The goal to define something that works for all implementations of distributed social networking is a hard one. In fact, I might even say it is impossible, but nonetheless, ActivityPub tried. If you combine ActivityPub and ActivityStreams, you will realize that these specifications offer you *a lot* of flexibility. With the amount of different actors, activities, objects, and the high amount of customizability defined within ActivityStreams, it should be possible to represent any kind of interaction possible. However, in my opinion, this creates more issues than it is trying to solve: the high number of possibilities makes it *very hard* to write server implementations and user interfaces able to handle this. I will go into details at a later point in this post.

It is also notable that ActivityPub is not only defining server-to-server communications. It

also covers the communication between servers and the client. This is a very interesting approach, and in theory, it would solve a lot of issues. Clients could be compatible with multiple server implementations, and extensive libraries to build software could be build. In reality, however, I expect a lot of ActivityPub implementations to use some kind of customized payloads, so clients might not be interchangeable at all.

Also, as I outlined earlier, there are a lot of different approaches to social networking. There are a lot of design considerations around how a network should look and feel, and the clients for these networks are designed around their core concepts. Based on ActivityPub, you can build applications that share concepts with Facebook, as well as applications more like Twitter. One could assume that all compatible clients work just fine with all ActivityPub implementers, allowing the user to choose their preferred style of network. However, I think ActivityPub offers too many ambiguous ways to implement certain aspects, so I doubt this would work.

Ultimately, I feel like server-to-client communication should not be a part of a specification that works on server-to-server communication. There are a lot of different requirements conflicting here, and while ActivityPub's approach is nice, a more complex server that provides a *simple* REST API for entities to clients would be the better approach, since building a client based on ActivityPub actually makes the job harder.

Profile discovery

Undeniably, a really important part of social networking is being able to find and get in touch with people. It is important to ensure that finding your contacts is not too hard. This task is easy for centralized systems: they already have all the profile data. You are easily able to search for names, locations, even phone numbers. In distributed systems… not so much. Since data is distributed over a lot of nodes, there is a high chance your node never received the profile of the person you want to contact.

This means contacts have to exchange some kind of identifier. For diaspora* (and, in fact, a lot of current systems), we use an account identifier that looks like an email address: `alice@example.com`, for example. This is a simple and obvious decision, since it includes all the relevant information for us: the username and the host. From there, we

use WebFinger and well-known URLs to discover details and fetch information.

With ActivityPub, this is not the case. Instead of having a fixed-format account identifier, users are actually much more like JSON documents floating around on some webserver. There is no fixed URL you can query for user information if you know the username, you have to know the full URL. This can be something simple like `https://example.com/alice/`, but obviously, this entirely depends on the server implementation, and can be more complex.

This is a design decision made by the people in the Social Working Group. Parts of the folks there *really* dislike non-dereferencable[2] identifiers, and their reasoning is something along the lines of "the entire web works with full URLs, so social networking should do, too". Regardless of the discussion about non-dereferencable identifiers, this was a bad decision. It does make the protocol easier, since developers do not have to write anything about discovering accounts, but it deeply impacts the user experience.

I am confident that, for most non-technical users, something that just looks like an email address is *way* easier to remember than remembering an arbitrary URL.

User profiles aka Actors

There is no such thing as a *profile*, really. But there is something called "actor". An actor is an object that can, well, act on other objects. Users are actors. There are several actor types: besides the to-be-expected `Person`, there is also `Application`, `Group`, `Organization`, and `Service`. I really like this approach, since it makes it easy to distinguish between persons, organization profiles, and bots.

All of these actors inherit their attributes from the generic ActivityStream `object`, so they have a lot of attributes. Besides some endpoints like `inbox` and `outbox`, which I will cover later, you can provide a name, a profile description, and some other details.

However, this list is not exhaustive. I immediately spot that a `birthday` field is missing, and here comes the first catch with ActivityStreams. Quoting the ActivityStreams specification here:

> This specification intentionally defines Actors in only the most generalized way, stopping short of defining semantically specific properties for each. All Actor objects are specializations of Object and inherit all of the core properties common to all Objects. External vocabularies can be used to express additional detail not covered by the Activity Vocabulary.

To be fair, in this case, they go on and mention implementations *should*[3] provide more profile details in a vCard. However, even vCard will not cover all imaginable use-cases, but because of the very open and extensible nature of ActivityStreams, you are free to supply any format you want. The handling of unknown attributes is not specified, but I assume implementations would simply ignore them, which could work fine in most cases. However, this is just one of the many sources of confusion. Just as an example, if you have a public profile description and a private profile description, there is no way to map that to the existing fields, so you end up defining custom fields.

Ultimately, you will end up with a lot of different custom fields for different use cases, and everyone is *kind of* cooking their own soup anyway. But because exchanging objects still works just fine, some missing fields can cause a lot of confusion. In my example, people would very easily get confused why their private profile description is visible on some servers but not on others.

Collections, Objects, and Activities

ActivityPub defines `Collections`, which are just a collection of objects. As such, there are two special collections for federating contents. There is the `outbox` and the `inbox`. If you want to send something to someone, you push an `Activity` to your `outbox` and the server takes care of sending it out. Likewise, all incoming activities will land in your `inbox`.

It should be no surprise that the URLs for these collections are not static, either. Instead, they are defined in the `Actor` object. This means you have to store the URLs for those in your database and use lots of different URLs, but that is not a big issue. Interestingly, this allows you to `GET` another actors `outbox`, effectively receiving a list of their past activities.

Just to make everything clear, I will explain two more important words. `Objects` are items you can pass around, like `Image`, `Note`, or `Profile`. `Activities` on the other hand are the *actions* an `Actor` does to an `Object`, like `Create`, `Follow`, `Like` and so on. The lists of available types are *really* long, so that is good. I am pretty sure that you can model just about any interaction you can have.

Servers never send `Objects` alone, they are always encapsulated in an `Activity`. This makes sense: a note does not just appear out of nowhere, someone has to `Create` it. Overall, I really like this concept.

There is an option to send public activities, which is a bit like broadcasting your post into the world. Since all activities require a recipient, you have to use a special address to mark a post as public.

To avoid having to send an object into everyones inbox, there is an optional inbox called the `sharedInbox`. The shared inbox is a bit comparable to a global receive URL per server... except it is not. Like with the other endpoints, there is no guaranteed URL, and there could be more than one shared inbox. Instead of having one inbox per server, each actor may specify a `sharedInbox` attribute. When delivering activities, you only have to send it once per `sharedInbox`, which may or may not be once per host. It is the receiving servers' responsibility to then route the item to individual inboxes. Also, the spec allows to simply hit all shared inboxes if an activity is marked as public.

Pulling/retrieving an activity is easy, since, as expected, their IDs are just URLs you can pull. Please note that, according to the specification, it is required that a server responds to these queries. However, requiring authentication for some or all interactions is *optional*. I will write a bit more about this later, but keep in mind that, in theory, a server would be perfectly spec-compliant with publicly distributing non-public activities.

The "X-Follows-Y" contact model

With all the flexibility in both ActivityPub and AcitivtyStreams, I was really surprised to see a really fixed relationship model between actors. Users follow each other. Here, this has several implications. When Bob is following Alice, Alice will send all public activities

(or all activities sent to all followers) to Bob, but Bob has no obligation to return something. While this model works fine for applications like Twitter, I do not think it is a good generic solution.

In diaspora*, we do the opposite. Alice can *start sharing* with Bob. From that point on, Bob's host will receive public posts from Alice. Alice has the option to now target Bob in *limited* (i.e. non-public) posts. Also, Alice will send her private profile to Bob, and Alice's server will fetch Bob's public profile[4]. In my opinion, the "specifying to whom I push content to" model is more fitting than the "specifying who should push me content" model, at least for distributed social networking. In any case, adopting ActivityPub in diaspora* would cause a complete rework of that logic, as well as potentially confusing all users.

I have to talk about collections again. Each `Actor` has two "contact collections", `following` and `followers`, again, with no fixed URLs, but fixed names. These can be used as a recipient collection for activities, to easily allow sending something to all of your followers.

I am not sure if a server can hold more collections. In diaspora*, we allow users to group contacts into what we call "aspects", and those aspects get used as a target for publishing content. In theory, this could be handled with having multiple collections. However, since the collection remains in the `to` field[5], this would be a bad solution for people who call their collections "annoying people" or "secret crush". This can probably solved by using `bto` or `bcc`, which have to be removed, leaving an empty `to` behind. However, even this is probably not a working solution, since replies are delivered based on the visible recipient fields. Directly setting everyone as `to`/`cc` would not solve this either, since the contacts would be visible again and an author might not want to show everyone who is also receiving the content.

All in all, I am surprised by the limited possibilities here, given that everything else in ActivityPub/ActivityStreams is designed to build literally everything.

Private messaging

Talking about missing things I am surprised about: there is no support for private communication. Most large implementation of distributed social networking have some form of private conversation, and given the usage I see on diaspora*, it seems like this is a required feature.

While private conversations are technically just objects, they get exposed in a different form of user interface. While regular content is usually shown in a "stream"-like fashion with comments underneath it, private conversations are strictly separated and show messages in a more linear, grouped, fashion.

One could technically implement private conversations by using activities with only one recipient, but distinguishing between regular content and private conversations, and building a different UI for that, would be tricky. The other solution would be to use custom activities with custom objects, but that would make interoperability a matter of luck, and nothing you can rely on.

General ambiguity in Objects and Activities

As mentioned earlier, there are a lot of objects and even more activities. While I am fine with the activities, some objects confuse me. There is `Article`, `Document`, and `Note`. Let's check their descriptions:

> Article - Represents any kind of multi-paragraph written work.
> Document - Represents a document of any kind.
> Note - Represents a short written work typically less than a single paragraph in length.

So... what should diaspora* use? If I have a look at the example, `Document` seems to be the wrong choice, since it does not include contents, but only a URL. However, all these objects inherit all their attributes from the generic `Object` type, so everything contains `content`, `url`, or both. Generally speaking, *most* posts on diaspora* are less than a single paragraph, but they might as well get really, really long.

Even worse: sometimes, people attach images to their posts. While every object also has

an `attachment` attribute, which is an array of objects to attach to, what if an user wants to be able to embed an image, a video, or a location at a very specific position?

If all networks were to agree on simply using `Note` for post-like activities, contents would still not be exchangeable without problems. The `content` attribute is one of those. While ActivityPub seems to assume the content is always HTML and thus added the `source` attribute for markup languages, this is not defined. In fact, the ActivityStreams spec is clear that it is *not* always HTML:

> By default, the value of content is HTML. The mediaType property can be used in the object to indicate a different content type.

So, it can be HTML, but it can also be something very different. diaspora* exclusively submits markdown, Friendica sends their BBCode by default. And since the ActivityPub specification is not limiting to HTML, this is another huge issue for interop.

ActivityStreams extensibility is another cause of concern. While it is *nice* to have a protocol that you can use for everything, having a specification that allows to send what basically is arbitrary JSON, is a huge danger for the compatibility between implementations. While the specification clearly warns that "implementations that rely too heavily on the use of extensions may experience reduced interoperability with other implementations", having such possibilities in the first place is maybe not a good idea if you *want* something that is interoperable.

Replies and Interactions

A reoccurring issue with distributed social networking is a high sensitivity to inconsistencies in the delivery of interactions (likes, comments, etc.). ActivityPub acknowledges this issue and writes the following:

> The following section is to mitigate the "ghost replies" problem which occasionally causes problems on federated networks. This problem is best demonstrated with an example.

> Alyssa makes a post about her having successfully presented a paper at a conference and sends it to her followers collection, which includes her friend Ben. Ben replies to Alyssa's message congratulating her and includes her followers collection on the recipients. However, Ben has no access to see the members of Alyssa's followers collection, so his server does not forward his messages to their inbox. Without the following mechanism, if Alyssa were then to reply to Ben, her followers would see Alyssa replying to Ben without having ever seen Ben interacting. This would be very confusing!

Yes, indeed, that would be very confusing.

Let us have a look at the implementation in the diaspora* protocol first. We have a pretty easy rule: Whenever Bob interacts to something Alice shared, that interaction will be sent to the Alice's host and the Alice's host alone. Since interactions are designed to be relayable without loosing the option to validate them, the Alice's host can forward these interactions to everyone who received the shareable in the first place. Alice's host is the one who delivered the shareable, so it feels somewhat natural to also ask Alice's host to distribute the interactions.

That is, however, not the case with ActivityPub. In the example (and, in fact, everywhere else), it is clear that Bob has to address individual recipients in his interaction. We have seen this implementation before and *it is built to fail*. Just imagine this very simple case: Alice posts an activity to Bob, but has Charlie on `bcc`. Bob posts a response. Bob sends his response back to Alice.

Since the `to` field of that response does not include a collection owned by Alice's host, the ActivityPub specification clearly says Alice's server *must not* forward this interaction. Because of that, Charlie will never receive Bob's response. However, when Alice then responds to Bob's comment, Charlie will get Alice's response, since Alice's server knows that Charlie is on `bcc`. The way of sending interactions outlined by ActivityPub does *not* solve the ghost reply issue. If anything, it creates more complex and confusing edge-cases, since some interactions will be forwarded, while others will not.

As per the ActivityStreams spec, all objects have a `replies` property. So, a more sensible, reliable, and even more AcitivityStream'y way of handling replies would probably

be adding the interaction to the replies collection and sending an `update`.

Authentication, Authorization, Verification, Encryption

I saved this point for the end, as it is both the most easy and the most difficult point in case of ActivityPub. Technically, all of the points in this sub-headline are not part of the specification, and thus, I could say "there is no such thing in ActivityPub". In reality, though, this is not so easy.

While it is true that these issues are not part of the official specification, there is a wiki page talking about authentication and authorization. These notes are, as I said, not part of any specification and are mere best practice reports. It is *not* safe to assume all implementation will follow those, and it is *not* safe to assume these are reviewed, or even stable. That being said, I still want to share my thoughts on them.

For client-to-server authentication, they suggest using OAuth 2.0. For working around the client registration on multiple endpoints, they suggest using OpenIDs Dynamic Client Registration. There is nothing bad to say about this; I really like it.

Now for the tricky part. As I mentioned earlier, some endpoints may allow accessing potentially private data. A good example for that is retrieving an activity from its URL. When you do not put any authentication in place, everyone can get access to any activity as long as the ID is known. This is not good.

In the wiki page, they suggest HTTP signatures for authorizing HTTP requests. While this is surely not the prettiest method, it is probably the most appropriate in this case, since ActivityPub highly depends on having everything accessible via plain HTTP anyway. Basically, if Alice wanted to fetch an activity from Bob, Alice's server would sign the HTTP request with Alice's private key, so Bob can validate that with the cached version of her public key.

As for actually signing the contents of an activity, not just the transport request, the Working Group suggests using Linked Data Signatures. This would work just fine and

indeed ensure that contents cannot be modified. But there is a catch. The working group claims that “for most forms of delivery, HTTP Signatures is enough”. I disagree.

In fact, I would love to see signatures being *required* everywhere. When distributing contents, for example as a payload in something like a *reshare*, it would be way too easy to alter the contents without a signature. If they, for whatever reason, do not want to have signatures, they could just stop sending payloads entirely, and just distribute IDs which the receiving hosts could then fetch.

Another important aspect, which is unfortunately completely missing, is encryption. Communication within ActivityPub *cannot be encrypted*. If you consider that the specs do not even require `https`, this is really bad. Although they *prefer* using `https`, the specification lists `http` as a valid protocol as well. So you might end up distributing potentially private contents unsigned and unencrypted, and you might not even be aware of it.

Distributed social networks are distributed for a reason, and privacy and data ownership are two of them, for sure. Signatures for all contents and encryption for private contents should be mandatory. They are mandatory in diaspora* since its beginning, and I cannot come up with a reason against that.

Summary

Okay, I’ll make this *short* after this huge post, I promise.

ActivityPub tries to work for everything and everyone. And because of that, they introduced a lot of flexibility and, sadly, a lot of ambiguity. Even though they tried, I found some reasons as for why we, as diaspora* developers, would not be able to build upon this new protocol without using heavily customized objects and activities.

Due to my past personal and professional[6] life, I do not like specifications and protocols that allow implementations to *do their own thing*. If you want your specification to be interoperable, you have to take care of that. Even the slightest wiggle room will cause issues sooner or later, and when you are dealing with peoples’ social networks, this causes

unsatisfied and confused users very quickly. This is not the way to grow.

In my personal opinion, a *very strict* specification would be the only way for me to be comfortable. And I am not even talking about limiting everyone to one set of fields and objects that everyone has to use. Instead, in my opinion, a specification should define all possible objects' types and all possible fields, define which attributes are optional and which are required, and rule out any room for interpretation or extension outside this specification. If an implementation needs additional fields to build the features they want, the specification maintainers should adapt the spec accordingly after some discussion and consideration.

I am aware that the W3C is not the right organization for such an approach. I do not expect to see a *Living Standards* in the W3C and maybe it is not fair to judge ActivityPub by its framework. They did a good job, and for sure, they are not able to frequently update the list of allowed fields, so I do not expect that. Still, I believe that a "Living Federation Protocol" would be the only way to build something that works for *most* folks, can be considered interoperable by default, and can quickly adapt to new requirements.

As much as I hope that ActivityPub and the related documents get traction and find widespread use, we, as the diaspora* project, and I personally, am neither comfortable nor *able* to support it. For diaspora*, it would bring a lot of downsides with no real upside. Even if we would migrate our federation, we would still be incompatible with a lot of stuff, and probably have a more inconsistent behavior altogether. And as for me, personally, I do not feel comfortable suggesting ActivityPub as a specification that ensures interoperability. Some of the core design decisions make everything way too undefined for me to be comfortable claiming so.

Update 2019-01-13: I published a *second part* of this post, *"ActivityPub - Final thoughts, one year later."*, with more thoughts on the general development, specifications, and other related topics.

Footnotes

1. One thing I should add: I did not contribute to the SocialWGs efforts, and thus to the specifications I am writing about, in any meaningful manner. While I did read the IRC channels, meeting minutes, and attended some meetings, at the point I was able to contribute, it was pretty clear that the goal of this working group was to build something based upon an improved version of ActivityStreams. Since I disagreed with this decision in the first place, me getting involved in the working groups efforts would actually have hampered their work. However, because of my lack of involvement, you are free to think my opinion is a bit hypocritical, and that is fine. ↩
2. I actually think that, although the `[user]@[host]` address does violate some Linked Data concepts, it is still pretty easy to look up. However, this is not a fight I want to be a part of. ↩
3. Note that, as usual with specs, this is an RFC 2119 SHOULD, which means, it is a recommendation, but not binding. You would be perfectly spec compliant with ignoring it. In fact, Mastodon, for example, does not provide a vCard. ↩
4. This profile exchange is mainly done for cryptographic reasons, since we use that to exchange the public keys. ↩
5. At least, this is how I understand the document. It's not written otherwise, so I assume so. ↩
6. My work includes figuring out why web things work in some browsers but not in others, and you would be surprised if I told you how creatively people can misunderstand specifications, even if there is no obvious room for that. ↩