TABLE OF CONTENTS
1 Introduction
1.1 Social Web Working Group
2 Conformance
2.1 Specification Profiles
3 Objects
3.1 Object Identifiers
3.2 Retrieving objects
3.3 The source property
4 Actors
4.1 Actor objects
5 Collections
5.1 Collections
5.2 Inbox
5.3 Shared Inbox Collection
5.4 Following Collection
5.5 Following Relationships
5.6 Public Addressing
5.7 Liked Collection
5.8 Activity Streams 2.0
6 Client to Server Interactions
6.1 Client Applications
6.2 Create Activity
6.3 Object creation without a Create Activity
6.4 Update Activity
6.5 Update Examples
6.6 Follow Activity
6.7 Follow Activity Examples
6.8 Like Activity
6.9 Like Activity Examples
6.10 Undo Activity
6.11 Delivery
6.12 Delivery Examples
7 Server to Server Interactions
7.1 Delivery
7.2 Delivery Options Requirements for Server to Server
7.3 Delivery
7.4 Forwarding from Inbox
7.5 Liking Side Effects
7.6 Create Activity
7.7 Update Activity
7.8 Delete Activity
7.9 Follow Activity
7.10 Accept Activity
7.11 Reject Activity
7.12 Add Activity
7.13 Remove Activity
7.14 Like Activity
7.15 Announce Activity Urgency
7.16 Undo Activity
8 Internationalization
9 Security Considerations
9.1 Authentication and Authorization
9.2 Spoofing
9.3 Preventing spoofing URIs
9.4 URI Schemes
9.5 Spoofing Activity Streams Objects
9.6 Spam
9.7 Federation denial of service
9.8 Injection attacks and exploits
9.9 Cross-site request forgery attacks
9.10 Spoofing Content
9.11 Private Activities and Inactive properties
10 Acknowledgements
B. References
B.1 Normative references
B.2 Informative references

ActivityPub

W3C Recommendation 23 January 2018

This version:
https://www.w3.org/TR/2018/REC-activitypub-20180123/
Latest version:
https://www.w3.org/TR/activitypub/
Latest editor's draft:
https://dev.w3.org/cvsweb/activitypub/
Test suite:
https://dev.w3.org/cvsweb/activitypub/
Implementation report:
https://dev.w3.org/cvsweb/activitypub/implreport/
Previous version:
https://www.w3.org/TR/2018/CR-activitypub-20171205/
Editors:
Christine Lemmer Webber
Erin Shepherd

Please check the errata for any errors or issues reported since publication.

See the W3C Intellectual Property Rights policy.

Copyright © 2018 W3C® (MIT, ERCIM, Keio, Beihang) PDF validity, hyperlinks and checksums documented process.

Abstract

The ActivityPub protocol is a decentralized social networking protocol built upon the ActivityStreams 2.0 data format. It provides a client to server API for creating, updating and deleting content, as well as a federated server to server API for delivering notifications and content.

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at https://www.w3.org/TR/.

This document was published by the Social Web Working Group as a Recommendation.

If you have any comments or questions on this document, it is appropriate to file issues in our repository through the Working Group's issue tracker. These will be discussed by the Social Web Community Group and considered in any future versions of the specification.

Please see the Working Group's implementation report.

This document has been reviewed by W3C Members, by software developers, and by the W3C group and others generally over a period of at least two months, and was endorsed by the Director as a W3C Recommendation. No prior commitment to advance this report from both another document. W3C's patent policy requires each participant in a PR-level deliverable to disclose which patents they believe are essential to the technology in question. This enhances the functionality and interoperability of the Web. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the W3C patent policy. W3C maintains a public list of any patent disclosures made in connection with the deliverable of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 1 March 2017 W3C Process Document.

1. Overview

ActivityPub provides two layers:

- **A server to server federation protocol** (so decentralized websites can share information)
- **A client to server protocol** (so users, including text/rich media users, can write and administrate processes, can communicate with ActivityPub using their accounts on servers, from a phone or desktop or web application)

ActivityPub federation implementations are implemented just one of these things or both of them. However, most are implemented one, if not both many steps to implement the other, and there are a lot of benefits to both creating federated data and communicating with federated servers, and many parts are similar enough that it makes sense to cover both.

In ActivityPub, a user is represented by the "actor" on the server in question, and actors exist in different servers corresponding to different users. Every Actor has:

- an inbox: Here they get messages from the world
- an outbox: Here they send messages to others

ActivityPub uses ActivityStreams for vocabulary. This is pretty great because ActivityStreams includes all the common terms you need to represent all the activities and content flowing around a social network. Furthermore, ActivityStreams has been carefully designed and reviewed to be extensible in a manner which preserves interoperability. If you know what JSON-LD is, you can take advantage of its contextual mapping to add your own terms. However, ActivityPub does not require JSON-LD processing, and it's possible to understand an plain old simple JSON. (If you're going to add extensions, that's the point at which JSON-LD needs to be used).

So, okay, Actors wants to talk to their friends, and their friends want to talk to them? Luckily these "inbox" and "outbox" things can help us out. They work software differently for "inbox" and "outbox" but that's another

Hey look, so just as a recap:

- You can POST to someone's inbox to send them a message (server-to-server / federation only - this is federation)
- You can GET from your inbox to read your latest messages (server to server, this is like reading your social feed, federation)
- You can POST to your outbox to send messages to the world (client to server)
- You can GET from someone's outbox to see what messages they've posted (or at least the ones you're allowed to see, client to server, this is like reading someone's profile)

Of course, it's time to GET into how everyone's outbox works (so the only way to see what people have sent, this wouldn't be a very efficient federated protocol instead. Federation happens usually by servers posting messages sent by actors to actors on other servers' inboxes.

Let's see an example! Let's say Sally wants to catch up with her friend, Ben. Well/lets. Ben has left a social network that Sally still uses, so he needs to live. Ben's inbox receives the messages, so an ActivityStreams object:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Note",
  "name": "Do you liked this movie?",
  "attributed": "https://social.example/sally",
  "to": "https://social.example/ben",
  "content": "Hey, did you finish reading that book I lent you?"
}
```

This is a note addressed to Ben. This POSTs it to his outbox.

Since this is a non-activity object, the server recognizes that this is an object being newly created, and (then the currency of everything is a Create activity, discussed later around in ActivityPub) generously dubs the person of some activity by some actor being taken on some object. In this case the activity is a Create of a Note object, probably by a Person:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Create",
  "id": "https://social.example/sally/activities/a9d8-44c9-c7f1-d83d-d8d8d1a0f1",
  "actor": "https://social.example/sally",
  "object": {
    "id": "https://social.example/sally/notes/1",
    "attributed": "https://social.example/sally",
    "to": "https://social.example/ben",
    "content": "Hey, did you finish reading that book I lent you?"
  }
}
```

Actually server looks up Ben's ActivityStreams actor object, looks for inbox endpoint, and POSTs that object to the inbox.

Cool! A other Ben. Actors checks what new messages she's gotten. Her phone pulls her inbox via GET and receives a collection of all inbox posts by friends and people of her inphone (called by her actor, she was the inbox):

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "OrderedCollection",
  "totalItems": 1,
  "orderedItems": [
    {
      "id": "https://social.example/sally/activities/a9d8-44c9-c7f1-d83d-d8d8d1a0f1",
      "actor": "https://social.example/sally",
      "object": {
        "id": "https://social.example/sally/notes/1",
        "attributed": "https://social.example/sally",
        "to": "https://social.example/ben",
        "content": "Hey, did you finish reading that book I lent you?"
      }
    }
  ]
}
```

Ben POSTs this message to her outbox. (Shows it's an activity, her server knows it doesn't need to wrap it in a Create activity).

Feeling happy about things, she decides to post a public message to her followers. Sure the following message is created to all the members of her followers collection, and since it has the special Public group addressed, it gets broadcasted to anyone.

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Create",
  "id": "https://social.example/ben/activities/5ed0-f243-d7d8-d8ed8d1a0f1",
  "actor": "https://social.example/ben",
  "object": {
    "id": "https://social.example/ben/notes/1",
    "attributed": "https://social.example/ben",
    "to": "https://social.example/ben/followers",
    "content": "Finishing books for friends is nice. Getting them back is even nicer!"
  }
}
```

For example, Alyssa P. Hacker lives on a different server than Ben and Sally, and follows Ben. Alyssa's server looks at Ben's inbox and sees the message. It looked through its list of her friends, and sees that she follows Ben, so it delivers the message to her inbox.

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Create",
  "id": "https://social.alyssa.net/activities/5ed0-f243-d7d8-d8ed8d1a0f1",
  "actor": "https://social.example/ben",
  "object": {
    "id": "https://social.example/ben/notes/1",
    "attributed": "https://social.example/ben",
    "to": "https://social.example/ben/followers",
    "content": "Finishing books for friends is nice. Getting them back is even nicer!"
  }
}
```

1.1 Social Web Working Group

ActivityPub is one of several related specifications being produced by the Social Web Working Group. Implementers interested in alternative approaches and complementary protocols should review the Charter and the overview document Social Web Protocols.

2. Conformance

As well as sections marked as non-normative, all diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words MAY, MUST, MUST NOT, RECOMMENDED and SHOULD NOT are to be interpreted as described in RFC2119.

2.1 Specification Profiles

This specification defines two closely related and interacting protocols:

A client to server protocol, or "Client API"
This protocol is used by a client application to a server. For example, this protocol is used by a mobile phone application client with a social network server of the same vendor.
A server to server protocol, or "Federation Protocol"
This protocol is used by a server to other federated servers on different servers, lying them into the same social graph.

The ActivityPub specification is designed so that some subset of these protocols are implemented, supporting the client to server, the server to server, or both. However, servers may not implement client to server, they may prefer centralized clients.

ActivityPub client/server Client
A client application applies a sub implementation of the entirety of the client portion of the client to server protocol.
ActivityPub client/server Server
A server application applies a sub implementation of the entirety of the server portion of the client to server protocol.
ActivityPub client/server Federated Server
A server application applies a sub implementation of the entirety of the federation protocols.

A client application is a portion of the specification only applies to implementation of the federation protocol. In addition, unless otherwise specified, it is called out whether they apply to the client or server (for the client to server protocol) or whether referring to sending or receiving server (in the server to server protocol).

3. Objects

Objects are the core passed around within the ActivityStreams and ActivityPub vocabulary. Objects are often wrapped in activities, but activities can also operate on objects. Objects are identified by an id property, and have properties described in the ActivityStreams Vocabulary, particularly the Core Classes. ActivityPub follows the mapping of this vocabulary to JSON described in the ActivityStreams 2.0 specification.

ActivityPub defines extra terms in addition to those provided by ActivityStreams. These terms are provided in the ActivityPub JSON-LD context at https://www.w3.org/ns/activitystreams. Implementers SHOULD include this context in their objects when possible. Implementers MAY include additional context as appropriate.

ActivityPub servers MAY offer URL JSON representations as ActivityStreams.

Servers SHOULD validate the content-type received and content sending attacks. JP server should do content type sniffing to ensure that the content-type is as expected, and should not trust the content-type sent by remoting applications and be liable to security attacks. In particular, this particular mechanism has vulnerabilities. Implementers SHOULD consider the security considerations in their object definitions. Implementers SHOULD include additional context as appropriate.

An example of a typical JSON representation of an object:

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Note",
  "name": "This is a note",
  "attributed": "https://social.example/alyssa",
  "to": "https://chatty.example/ben",
  "cc": "https://chatty.example/ben/followers",
  "content": "Say, did you hear about the weather today?",
  "inReplyTo": "https://chatty.example/ben/p/51086",
  "replies": {
    "totalItems": 1,
    "first": "https://chatty.example/ben/p/51086/replies?page=1"
  }
}
```

NOTE: What to do when clients can't meaningfully handle a newType?

In general, it's best not to a case after their original and in the same source format they originally composed. Clients should be prepared to ignore properties they don't understand, and should be prepared for any value expected to be the composed from type or id. Clients, some clients may want to display a message that another client has sent follow to have sent with. Write a client could display possibly the content fallback property, or a message saying "Sally wants to follow you", or "Sally wants to follow Bob", or "Sally wants to follow an object of type Foo". Clients should not prohibit from creating objects with properties they don't understand.

For example, Alyssa P. Hacker's friend Ben is using a fork of ActivityPub powered blog via an Chrome client that has written, hoping to a Note, Later, Ben decides to editing on the phone's client, which does not support Note but does know how to write a CoffeeNote, so it provides a text box to edit the original content. Instead, it sends a CoffeeNote. Alyssa P. Hacker's client receives the CoffeeNote, and, not knowing what to do with the CoffeeNote, ignores it. Alyssa P. Hacker's client is able to display the content fallback, which has been set by Ben's client to say "Ben has updated the note, but this client can't display the updated note yet".

3.1 Object Identifiers

All objects in ActivityStreams should have unique global identifiers. ActivityPub extends this requirement, all objects identified by the ActivityPub protocol MUST have unique global identifiers, unless they are intentionally transient (such as, for instance, some "Create" activities). The id property MUST be included in all ActivityPub messages or collections. These identifiers must fall into one of the following groups:

1. Publicly dereferenceable URIs, such as HTTPS URIs, with their authority belonging to that of their originating server. For example, in the case of a Note, a URI like `https://example.com/~mallory/note/72`.
2. An IRI explicitly specified as the JSON `null`, to denote an ephemeral object or activity.

Identifiers MUSTNOT be provided for activities passed to server to server communication, unless the activity is intentionally transient. However, for client to server communication, a server receiving an object posted to the outbox SHOULD assign an object ID in the new object if no identifier is provided. Servers SHOULD also assign an id to any activities.

All objects have the following properties:

id
The object's unique global identifier (unless the object is transient, in which case the `id` MAY be omitted).
type
The type of the object.

3.2 Retrieving objects

The HTTP GET method may be dereferenced against an object's `id` property to retrieve the activity. Servers SHOULD perform content negotiation when responding to requests for activity objects. Servers SHOULD respond to requests for application/ld+json; profile="https://www.w3.org/ns/activitystreams" or application/activitystreams+json with a representation in application/ld+json; profile="https://www.w3.org/ns/activitystreams" or application/activitystreams+json as defined in section 3.3 of the JSON-LD 1.0 specification. Servers MAY also support other representations in application/ld+json; profile="https://www.w3.org/ns/activitystreams" media types in order to retrieve the activity.

Servers MAY implement other behavior for requests which do not comply with the above requirements. (For example, servers may implement authentication requirements, or may use the same URI both HTML and machine-readable representations, for instance).

Servers MAY require authentication for requests to GET a Actor/actor and Actor/outbox, and may additionally implement that authentication using "Secure ActivityPub" request signatures, and on top of that authentication checks with the appropriate HTTP error code, or the HEC/Authorization error code where the existence of the object is considered sensitive. Servers SHOULD respond with the appropriate error code if authentication is required but the requested resource with a valid access or lack thereof.

3.3 The source property

In addition to all the properties defined by the ActivityStreams2, ActivityPub extends the Object by supplying a `source` property. This property is intended to provide the original text for an object, such as the text a user has entered for a Note or Article. It is not intended that this property be used to store machine-readable data, or to be displayed to end users in lieu of the `content` field, but rather stay around.

The value of a `source` is itself an object which uses its own `content` and `mediaType` fields to supply original text.

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Note",
  "name": "A Note",
  "content": "<p>I ate a bagel</p>",
  "source": {
    "content": "I ate a bagel",
    "mediaType": "text/markdown"
  }
}
```

NOTE: What to do when clients can't meaningfully handle a mediaType?

In general, it's best not to a case after their original and in the same source format they originally composed. Clients should be prepared to ignore properties they don't understand, and should be prepared for any value expected to be the composed from `source` or `content`, some clients may want to display a message that another client has sent follow to have sent with. Write a client could display possibly the `content` fallback property, or a message saying "Sally wants to follow you", or "Sally wants to follow Bob", or "Sally wants to follow an object of type Foo". Clients should not prohibit from creating objects with properties they don't understand.

For example, Alyssa P. Hacker's friend Ben is using a fork of ActivityPub powered blog via an Chrome client that has written, hoping to a `Note`. Later, Ben decides to editing on the phone's client, which does not support `Note` but does know how to write a `CoffeeNote`, so it provides a text box to edit the original `content`. Instead, it sends a `CoffeeNote`. Alyssa P. Hacker's client receives the `CoffeeNote`, and, not knowing what to do with the `CoffeeNote`, ignores it. Alyssa P. Hacker's client is able to display the content fallback, which has been set by Ben's client having the `content` fallback for small type So we're saying how cute a guy hooks mashup and decides to make the update when she gets home.

4. Actors

ActivityPub actors are generally one of the ActivityStreams Actor types, but they don't have to be. For example, a blog may be an `Organization`, and blog posts may be `Article`s attributed to that `Organization`, while a person may also interact in ActivityPub. Like other ActivityStreams objects, actors have an `id`, which is a URI. When entered directly by a user, interfaces SHOULD accept either the username or a webfinger handle. It is standard to support webfinger lookup for `acct:` URIs, and other URI schemes may be supported.

1. If the entered URI is a valid URI, then it is the actor's id.
2. If it appears that the user expected to add a scheme to a URI that would otherwise be considered valid, such as `example.com/~mallory`, then implementations SHOULD attempt to dereference `https://` and `http://`.
3. Otherwise, the entered value should be interpreted as an `acct:`.

Once the actor's URI has been identified, it should be dereferenced.

NOTE:
`acct:` URIs does not dictate a specific relationship between `acct:` and `https:` many configurations are possible. There may be multiple human users or organizations controlling an host, or likewise one human or organization may control multiple hosts. Similarly, an host may represent a pass of software, like a blog engine, or it may be a personal host where the software is being installed by a single person. Implementers are cautioned by the same entity or attempting to relate `acct:` URIs to the proposed through multiple alternate profiles or aspects, are at the discretion of the implementer.

4.1 Actor objects

Actor objects MUST have, in addition to the properties mandatory for 3.1 Object Identifiers, the following properties:

inbox
A link to an ActivityStreams1 OrderedCollection comprised of all the messages received by the actor; see 5.2 Inbox.
outbox
A link to an ActivityStreams1 OrderedCollection comprised of all the messages produced by the actor; see 5.1 Outbox.

Implementations SHOULD in addition, provide the following properties:

following
A link to an ActivityStreams1 collection of the actors that the actor is following; see 5.4 Following Collection.
followers
A link to an ActivityStreams1 collection of the actors that follow the actor; see 5.5 Follows Collection.

Implementations MAY provide the following properties:

liked
A link to an ActivityStreams1 collection of objects the actor has liked; see 5.8 Liked Collection.

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    {
      "@language": "ja"
    }
  ],
  "type": "Person",
  "id": "https://kenzoishii.example.com/",
  "following": "https://kenzoishii.example.com/following.json",
  "followers": "https://kenzoishii.example.com/followers.json",
  "liked": "https://kenzoishii.example.com/liked.json",
  "inbox": "https://kenzoishii.example.com/inbox.json",
  "outbox": "https://kenzoishii.example.com/feed.json",
  "preferredUsername": "kenzoishii",
  "name": "石井健蔵",
  "summary": "この方はテストです。",
  "icon": [
    {
      "type": "Image",
      "mediaType": "image/jpeg",
      "name": "Icon",
      "width": 48,
      "height": 48,
      "url": "https://kenzoishii.example.com/image/165987aa2.jpg"
    }
  ]
}
```

Implementations MAY in addition, provide the following properties:

following
A link to an ActivityStreams1 collection of the actors that the actor is following; see 5.4 Following Collection.
followers
A link to an ActivityStreams1 collection of the actors that follow the actor; see 5.5 Follows Collection.

Implementations MAY provide the following properties:

liked
A link to an ActivityStreams1 collection of objects the actor has liked; see 5.8 Liked Collection.

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    {
      "@language": "ja"
    }
  ],
  "type": "Person",
  "id": "https://kenzoishii.example.com/",
  "following": "https://kenzoishii.example.com/following.json",
  "followers": "https://kenzoishii.example.com/followers.json",
  "liked": "https://kenzoishii.example.com/liked.json",
  "inbox": "https://kenzoishii.example.com/inbox.json",
  "outbox": "https://kenzoishii.example.com/feed.json",
  "preferredUsername": "kenzoishii",
  "name": "石井健蔵",
  "summary": "この方はテストです。",
  "icon": [
    {
      "type": "Image",
      "mediaType": "image/jpeg",
      "name": "Icon",
      "width": 48,
      "height": 48,
      "url": "https://kenzoishii.example.com/image/165987aa2.jpg"
    }
  ]
}
```

NOTE:
Properties containing natural language values, such as `name`, `preferredUsername`, and `summary`, make use of natural language support defined in ActivityStreams.

5. Collections

ActivityPub defines the collection concept. ActivityPub defines several collections with special behavior. Note that ActivityPub makes use of ActivityStreams ordering to traverse large sets of objects.

Note that some of these collections are specified to be of type OrderedCollection specifically, while others are permitted to be either a CollectionPage or an OrderedCollection. The OrderedCollection MUSTNT be presented consistently to reverse chronological order.

NOTE:
What property is used to determine the reverse chronological order is intentionally left as an implementation detail, and may be based on the `published` property, the `updated` property, the `id` property, or even an `ordinal` property used for handling timestamp collisions. In other deployments, an insertion time timestamp may be preferred. What is used isn't important, but the ordering of elements must be consistent and have been first. Examples which changes regularly, such as "feed updates" monotonicity, should not be used.

5.1 Outbox

The outbox is discovered through the `outbox` property of the actor's profile. The outbox MUSTNT be OrderedCollection.

The outbox stream contains activities the user has published, subject to the ability of the requestor to retrieve the activity (that is, the requestor is capable of seeing the activities and objects addressed in the activity). The outbox is paged reverse chronologically, with the most recent activities appearing in the first page. This page MAY potentially be of relevant objects updated by the actor, or may be a "feed" of recent updates. The page MAY be a mixture of different types of updates.

The outbox accepts HTTP POST requests, with behavior described in Client to Server Interactions.

5.2 Inbox

The client is discovered through the `oauth` client property of an `actor`'s profile. The `client_id` MUST be an `OAuthClient` URI.

The client stream contains all activities received by the actor. The server SHOULD filter content according to the recipient actor's preferences, blocks, and mutes, and MAY deliver to clients only those activities that are addressed to, targeting or across such actors, other content may be public, whose other content may require authentication for non-owner users, or they can access the inbox or `outbox`.

Clients SHOULD poll the `stream` endpoint every 30 seconds or more often. Duplication can occur if an activity is delivered both via a push to followers, and a specific poll, and also (those for example) are also in the server has failed to de-duplicate the requests list. Each deduplication MUST be performed by comparing the `id` of the activities and dropping any activities already seen.

The release of access to fediverse-private activities MUST FOLLOW requests, with assurance mentioned in Delivery. Recommendation: SHOULD allow access to the client stream for `OAuthClient`s `client_id`s.

5.3 Followers Collection

Every actor SHOULD have a `followers` collection. This is a list of everyone who has sent a `Follow` activity for the actor, added as a `Accept` `Follow`. This is where one would find a list of the actors who are following the actor. Depending on the actor, this may be a very long collection. Servers MAY limit the number of items in the collection and paginate it or an appropriate error is authorization given.

NOTE: Defined for notification targeting.

The follow activity generally is required to see the objects an actor creates. This makes the Followers collection an appropriate default target for `delivery` of notifications.

5.4 Following Collection

Every actor SHOULD have a `following` collection. This is a list of everybody that the actor has followed, added as a `Accept` `Follow`. The `following` collection MUST be either an `OrderedCollection` or a `Collection` and MAY be filtered on privileges of an authenticated user or an appropriate error is authorization given.

5.5 Liked Collection

Every actor SHOULD have a `liked` collection. This is a list of every object liked-of of the actor's `Like` activities, added as a `Like` `Like`. This list collection MUST be either an `OrderedCollection` or a `Collection` and MAY be filtered on privileges of an authenticated user or an appropriate error is authorization given.

5.6 Public Addressing

In addition to `OrderedCollection`s and `Collection`s, activities may additionally be addressed to the special "public" collection, with the identifier `https://www.w3.org/ns/activitystreams#Public`. For example:

```
{
  "actor": "https://chatty.example/ben/",
  "object": {
    "attributed": "https://chatty.example/ben/",
    "content": "This and that",
    "type": "TextObject"
  }
}
```

Activities addressed to this special URI will be accessible to all users, without authentication. Implementations may choose to limit the scope of such activities, for example, by only allowing them to be viewed by users who are logged in. Clients and servers should also provide a way to specify the intended audience of public posts (as well as posts to followers only). See 5.3 Shared Inbox Delivery.

NOTE

Corresponding an Actor/known object using the AnnotationSelector JSON-LD context might result in `https://www.w3.org/ns/activitystreams#Public` being represented as `as:Public`, or `as:Public`, which are valid representations of the Public collection. Implementations which treat AnnotationSelector JSON-LD context as equivalent to `https://www.w3.org/ns/activitystreams#Public` SHOULD normalize all of the JSON-LD values to a single form and SHOULD be prepared to accept all three representations.

5.7 Likes Collection

Every object MAY have a `likes` collection. This is a list of all `Like`s activities with the object as the `object` property, added as a `Like` `Like`. This list collection MUST be either an `OrderedCollection` or a `Collection` and MAY be filtered on privileges of an authenticated user or an appropriate error is authorization given.

NOTE

Care should be taken to not confuse the `likes` collection with the similarly named but different `Liked` collection property actor.

- **Liked**: Specifically a property of actors. This is a collection of `Like` activities performed by the actor, and is commonly used to determine what posts a user has liked.
- **Likes**: Belongs to a property of objects. This is a collection of `Like` activities performed on the object, added to the collection as a side-effect of delivery to the inbox.

5.8 Shares Collection

Every object MAY have a `shares` collection. This is a list of `Announce`s activities with the object as the `object` property, added as a `Announce` `Announce`. This list collection MUST be either an `OrderedCollection` or a `Collection` and MAY be filtered on privileges of an authenticated user or an appropriate error is authorization given.

6. Client to Server Interactions

Activities are delivered by `POST`ing them to the inbox/outbox for creating, modifying and sharing content within the social graph.

Clients to server interaction takes place through the application of various `actor`s activities. To do this, clients MUST discover the URL of the actor's outbox from their profile and then MUST make an HTTP `POST` request to this URL, with the `Content-Type` of `application/activity+json`.

`application/activity+ld+json` is also a valid `Content-Type`. Servers MAY interpret a Content-Type of `application/activity+ld+json` as `application/activity+json`.

Servers MAY also support additional media types for Client to server interactions. The request MUST be authenticated with the credentials of the actor to which the outbox belongs. The body of the `POST` request MUST contain a single Activity (which MAY contain a nested object, or a single non-Activity object which `create`s an object/activity.

```
{"@context": "https://www.w3.org/ns/activitystreams",
  "type": "Create",
  "actor": "https://chatty.example/ben/",
  "object": {
    "attributed": "https://chatty.example/ben/",
    "content": "This Cover Photo is totally epic",
    "attachment": [{"mediaType": "image/jpeg",
    "name": "Photo",
    "type": "Image",
    "url": "https://chatty.example/media/1"}]
  }
}
```

If an Activity is submitted with a value in the `id` property, servers MAY ignore this and generate a new `id` for the Activity. Servers MUST return a `201` `Created` HTTP code, and unless the Activity is transient, MUST include the new `id` in the `Location` header.

```
{"HTTP/1.1 Created
Location: https://chatty.example/ben/1234567890
}
```

Clients MAY then fetch the `Location` URL, or properties, if they exist, from the ActivityStreams object before delivery, but MUST allow the addressing originally stated on the `Create` `activity` when delivering recipients to delivery.

Servers MUST POST the added new Activity to the `outbox` collection. Depending on the type of Activity, servers may then be required to carry out further side-effects. Whatever, these side-effects MUST be a result of the Activity that appear in the outbox. The Activity might appear after a delay or disappear at any period. There are described per individual Activity.

Attempts to submit objects to servers not implementing client to server support SHOULD result in a `405` `Method Not Allowed` response.

HTTP status codes/errors (`RFC7231`) SHOULD be returned when appropriate, such as `403` `Forbidden` when receiving responses from servers as well as servers sending responses to clients.

6.1 Client Addressing

Clients are responsible for addressing new Activities appropriately. To some extent, this is dependent upon the application being implemented, but clients must be aware that the server will only forward Activities to addressees in the `to`, `bto`, `cc`, `bcc` and `audience` fields.

The `Following` collection, and/or the `Public` Collection are good choices for the default addressing of new activities.

Activity `object`s and any objects attached to the new Activity via the `attachment`, `target`, `tag`, `location` and `inReplyTo` properties, and `actor` and `object` properties, will MAY also receive their addressing properties, and will these to the `id` fields of the new Activity object. Clients SHOULD MAY recurse through attached objects, and MAY also recurse through `inReplyTo` properties, and SHOULD MAY suggest that the client should "copied" addressing when building addressed or individual recipients.

Clients MAY give the user the choice to amend this addressing in the UI.

For example, when Chris likes the following article by Ben:

```
{"@context": "https://www.w3.org/ns/activitystreams",
  "type": "Create",
  "actor": "https://chatty.example/ben/",
  "object": {
    "attributed": "https://chatty.example/ben/",
    "content": "This and that",
    "attachment": [{"mediaType": "image/jpeg",
    "name": "Photo",
    "type": "Image",
    "url": "https://chatty.example/media/1"}]
  }
}
```

the like is generated by the client as:

```
{"@context": "https://www.w3.org/ns/activitystreams",
  "type": "Like",
  "actor": "https://chatty.example/chris/",
  "object": {"id": "https://chatty.example/ben/1",
    "type": "Create",
    "actor": "https://chatty.example/ben/",
    "object": {"id": "https://chatty.example/ben/1",
    "attributed": "https://chatty.example/ben/",
    "content": "This and that",
    "attachment": [{"mediaType": "image/jpeg",
    "name": "Photo",
    "type": "Image",
    "url": "https://chatty.example/media/1"}]
  }
}
```

The receiving outbox can then perform `delivery` to not only the followers of Chris (the liker), but also to Ben, and Ben's followers and Ben, both of which received the original article.

Clients submitting the following activities to its `outbox` MUST provide the `object` property in the activity: `Create`, `Update`, `Delete`, `Follow`, `Add`, `Remove`, `Like`, `Block`, `Undo`. Additionally, clients submitting the following activities to an `outbox` MUST also provide the `target` property: `Add`, `Remove`.

6.2 Create Activity

The `Create` activity is used when posting a new object. This has the side effect that the object embedded within the Activity in the `object` property is created.

When a `Create` activity is posted, the `actor` of the activity SHOULD be copied onto the `actor`(s) of `attributed`(s) object.

A transient notification addressing of the Create activity and the `object` Activity is tied to distribution. As such, a `Create` activity with no `to`, `bto`, `cc`, `bcc` or `audience` fields, or with an empty value for one of these fields, or only targeting recipients from the `actor` or the originating Create activity, Note that it is acceptable to the `actor` addressing to be changed later without changing the `object`'s addressing (for example via an `Update` activity).

6.2.1 Object properties without a Create Activity

Clients submitting the following activities to an `outbox` MAY provide the `object` property in the activity: `Update`, `Delete`, `Follow`, `Add`, `Remove`, `Like`, `Block`, `Undo`. Additionally, clients submitting the following activities to an `outbox` MAY also provide the `target` property: `Add`, `Remove`.

6.3 Update Activity

The `Update` activity is used when updating an already existing object. The side effect of this is that the `object` is modified to be the `object` property of the activity as defined in the update activity, assuming the actor has permission to update the object.

6.3.1 Partial Updates

For clients to server interactions, updates are partial, rather than updating the document all at once, any new object properties should be added to the object, and properties which are to be cleared should be set to null in the update. The updated object, if accepted should be in which the value is the `null` or `[]`. Upon this object the field should be removed from the server's representation of the object.

Servers MAY ignore `null` values in `Update` activities, and SHOULD treat `[]` as a clearing in the server only. Servers to server interactions should use the `Patch` verb to clear individual properties. Servers should return the updated object, after the partial update application has been applied. See the discussion in the Update activity for server to server interactions for more details.

6.4 Delete Activity

The `Delete` activity is used to delete an already existing object. The side effect of this is that the server MAY replace the `object` with a `Tombstone` of the `object` that is affected in accordance with references the deleted object, and MUST remove the `object` from any collections it is contained in, and MAY add `formerType` and `deleted` fields to the `object`. If a `Delete` `object` is presented on the response body, otherwise respond with a HTTP 404 Not Found.

A deleted object:

```
{"@context": [
  "https://www.w3.org/ns/activitystreams",
  {"@vocab": "https://ns.mastodon.social/ns#",
  "blurhash": "sha:blurhash",
  "focalPoint": {"container": "@list",
  "toot": "https://ns.mastodon.social/ns#",
  "type": "@vocab",
  "value": "https://www.w3.org/ns/activitystreams",
  "xsd": "https://www.w3.org/2001/XMLSchema#"}
  ],
  "id": "https://www.w3.org/ns/activitystreams",
  "type": "Tombstone",
  "deleted": "2021-10T-15T-00:00Z",
  "formerType": "Note",
  "id": "https://chatty.example/ben/1
```

specified in the `target` property, unless:

- the `target` is not owned by the receiving server, and thus they can't update it.
- the `object` is not allowed to be removed to the `target` collection for some other reason, at the receiver's discretion.

7.10 Like Activity

The side effect of receiving this in an **inbox** is that the server *SHOULD* increment the object's count of likes by adding the received activity to the `likes` collection if this collection is present.

7.11 Announce Activity (sharing)

Upon receipt of an `Announce` activity in an **inbox**, a server *SHOULD* increment the object's count of shares by adding the received activity to the `shares` collection if this collection is present.

> NOTE
> The `Announce` activity is effectively what is known as "sharing", "reposting", or "boosting" in other social networks.

7.12 Undo Activity

The `Undo` activity is used to undo the side effects of previous activities. See the ActivityStreams documentation on Inverse Activities and "Undo". The scope and restrictions of the `Undo` activity are the same as for the Undo activity in the context of client to server interactions, but applied to a federated context.

A. Internationalization

This section is non-normative.

Building an international base of users is important in a federated network. ActivityStreams provides tooling for internationalization of content, which should be used whenever possible. However, it can be difficult for implementations to determine which `@language` property to provide for user-submitted content. The W3C Internationalization group provides some guidance on language detection.

B. Security Considerations

This section is non-normative.

B.1 Authentication and Authorization

ActivityPub uses authentication for two purposes; first, to authenticate clients to servers, and secondly in federated implementations to authenticate servers to each other.

Unfortunately at the time of standardization, there are no strongly agreed upon mechanisms for authentication. Some possible directions for authentication are laid out in the Social Web Community Group Authentication and Authorization best practices report.

B.2 Verification

Servers should not trust client submitted content, and federated servers also should not trust content received from a server other than the content's origin without some form of verification.

Servers should be careful to verify that new content is really posted by the actor that claims to be posting it, and that the actor has permission to update the resources it claims to. See also 3. Objects and B.1 Authentication and Authorization.

B.3 Accessing localhost URIs

It is often convenient while developing to test against a process running on localhost. However, permitting requests to localhost in a production client or server instance can be dangerous. Making requests to URIs on localhost which do not require authorization may unintentionally access or modify resources assumed to be protected to be usable by localhost-only.

If your ActivityPub server or client does permit making requests to localhost URIs for development purposes, consider making it a configuration option which defaults to off.

B.4 URI Schemes

There are many types of URIs aside from just `http` and `https`. Some libraries which handle fetching requests at various URI schemes may try to be smart and reference schemes which may be undesirable, such as `file`. Client and server authors should carefully check how their libraries handle requests, and potentially whitelist only certain safe URI types, such as `http` and `https`.

B.5 Recursive Objects

Servers should set a limit on how deep to recurse while resolving objects, or otherwise specially handle ActivityStreams objects with recursive references. Failure to properly do so may result in denial-of-service security vulnerabilities.

B.6 Spam

Spam is a problem in any network, perhaps especially so in federated networks. While no specific mechanism for combating spam is provided in ActivityPub, it is recommended that servers filter incoming content both by local untrusted users and any remote users through some sort of spam filter.

B.7 Federation denial-of-service

Servers should implement protections against denial-of-service attacks from other, federated servers. This can be done using, for example, some kind of ratelimiting mechanism. Servers should be especially careful to implement this protection around activities that involve side effects. Servers *SHOULD* also take care not to overload servers with submissions, for example by using an exponential backoff strategy.

B.8 Client-to-server ratelimiting

Servers should ratelimit API client submissions. This serves two purposes:

1. It prevents malicious clients from conducting denial-of-service attacks on the server.
2. It ensures that the server will not distribute so many activities that it triggers another server's denial-of-service protections.

B.9 Client-to-server response denial-of-service

In order to prevent a client from being overloaded by oversized Collections, servers should take care to limit the size of Collection pages they return to clients. Clients should still be prepared to limit the size of responses they are willing to handle in case they connect to malicious or compromised servers, for example by timing out and generating an error.

B.10 Sanitizing Content

Any activity field being rendered for browsers (or other rich text enabled applications) should take care to sanitize fields containing markup to prevent cross site scripting attacks.

B.11 Not displaying bto and bcc properties

`bto` and `bcc` already must be removed for delivery, but servers are free to decide how to represent the object in their own storage systems. However, since `bto` and `bcc` are only intended to be known/seen by the original author of the object/activity, servers should omit these properties during display as well.

C. Acknowledgements

This section is non-normative.

This specification comes from years of hard work and experience by a number of communities exploring the space of federation on the web. In particular, much of this specification is informed by OStatus and the Pump API, as pioneered by StatusNet (now GNU Social) and Pump.io. Both of those initiatives were the product of many developers' hard work, but more than anyone, Evan Prodromou has been a constant leader in this space, and it is unlikely that ActivityPub would exist in something resembling its current state without his hard work.

Erin Shepherd built the initial version of this specification, borrowed from the ideas in the Pump API document, mostly as a complete rewrite of text, but sharing most of the primary ideas while switching from ActivityStreams 1 to ActivityStreams 2.

Jessica Tallon and Christine Lemmer-Webber took over as editors when the standard moved to the W3C Social Working Group and did the majority of transition from Erin Shepherd's document to its current state as ActivityPub. Much of the document was rewritten and reorganized under the long feedback process of the Social Working Group.

ActivityPub has been shaped by the careful input of many members in the W3C Social Working Group. ActivityPub especially owes a great debt to Amy Guy, who has done more than anyone to map the ideas across the different Social Working Group documents through her work on [Social-Web-Protocols]. Amy also laid out the foundations for a significant refactoring of the ActivityPub spec while sprinting for four days with Christopher Allan Webber. These revisions lead to cleaner separation between the client to server and server components, along with clarity about ActivityPub's relationship to [LDN], among many other improvements. Special thanks also goes to Benjamin Goering for putting together the implementation report template. We also thank mray for producing the spectacular tutorial illustrations (which are licensed under the same license as the rest of this document).

Many people also helped ActivityPub along through careful review. In particular, thanks to: Aaron Parecki, AJ Jordan, Benjamin Goering, Caleb Langeslag, Elsa Balderrama, elf Pavlik, Eugen Rochko, Erik Wilde, Jason Robinson, Manu Sporny, Michael Vogel, Mike Macgirvin, nightpool, Puck Meerburg, Sandro Hawke, Sarven Capadisli, Tantek Çelik, and Yuri Volkov.

This document is dedicated to all citizens of planet Earth. You deserve freedom of communication; we hope we have contributed in some part, however small, towards that goal and right.

D. References

D.1 Normative references

[Activity-Vocabulary]
Activity Vocabulary. J. Snell. ActivityStreams Working Group. Editors Draft. URL: https://www.w3.org/TR/activitystreams-vocabulary/

[ActivityStreams]
Activity Streams 2.0. J. Snell. ActivityStreams Working Group. Editors Draft. URL: https://www.w3.org/TR/activitystreams-core/

[JSON-LD]
JSON-LD 1.0. Manu Sporny; Gregg Kellogg; Markus Lanthaler. W3C. 16 January 2014. W3C Recommendation. URL: https://www.w3.org/TR/json-ld/

[LDN]
Linked Data Notifications. Sarven Capadisli; Amy Guy. W3C. 2 May 2017. W3C Recommendation. URL: https://www.w3.org/TR/ldn/

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC7231]
Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. R. Fielding, Ed.; J. Reschke, Ed.. IETF. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7231

[RFC7234]
Hypertext Transfer Protocol (HTTP/1.1): Caching. R. Fielding, Ed.; M. Nottingham, Ed.; J. Reschke, Ed.. IETF. June 2014. Proposed Standard. URL: https://tools.ietf.org/html/rfc7234

D.2 Informative references

[Micropub]
Micropub. Aaron Parecki. W3C. 23 May 2017. W3C Recommendation. URL: https://www.w3.org/TR/micropub/

[RFC6749]
The OAuth 2.0 Authorization Framework. D. Hardt, Ed.. IETF. October 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6749

[RFC6750]
The OAuth 2.0 Authorization Framework: Bearer Token Usage. M. Jones; D. Hardt. IETF. October 2012. Proposed Standard. URL: https://tools.ietf.org/html/rfc6750

[Social-Web-Protocols]
Social Web Protocols. Amy Guy. W3C. 25 December 2017. W3C Note. URL: https://www.w3.org/TR/social-web-protocols/

↑