GNAP
Internet-Draft
Intended status: Standards Track
Expires: 13 January 2022

J. Richer, Ed.
Bespoke Engineering
A. Parecki
Okta
F. Imbault
acert.io

12 July 2021

Grant Negotiation and Authorization Protocol
draft-ietf-gnap-core-protocol-06

Abstract

GNAP defines a mechanism for delegating authorization to a piece of software, and conveying that delegation to the software. This delegation can include access to a set of APIs as well as information passed directly to the software.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 13 January 2022.

Copyright Notice

Table of Contents

1. Introduction

This protocol allows a piece of software, the client instance, to request delegated authorization to resource servers and to request direct information. This delegation is facilitated by an authorization server usually on behalf of a resource owner. The end-user operating the software may interact with the authorization server to authenticate, provide consent, and authorize the request.

The process by which the delegation happens is known as a grant, and GNAP allows for the negotiation of the grant process over time by multiple parties acting in distinct roles.

This specification focuses on the portions of the delegation process facing the client instance. In particular, this specification defines interoperable methods for a client instance to request, negotiate, and receive access to information facilitated by the authorization server. This specification also discusses discovery mechanisms for the client instance to configure itself dynamically. The means for an authorization server and resource server to interoperate are discussed in the companion document, [I-D.draft-ietf-gnap-resource-servers].

The focus of this protocol is to provide interoperability between the different parties acting in each role, and is not to specify implementation details of each. Where appropriate, GNAP may make recommendations about internal implementation details, but these recommendations are to ensure the security of the overall deployment rather than to be prescriptive in the implementation.

This protocol solves many of the same use cases as OAuth 2.0 [RFC6749], OpenID Connect [OIDC], and the family of protocols that have grown up around that ecosystem. However, GNAP is not an extension of OAuth 2.0 and is not intended to be directly compatible with OAuth 2.0. GNAP seeks to provide functionality and solve use cases that OAuth 2.0 cannot easily or cleanly address. Appendix B further details the protocol rationale compared to OAuth 2.0. GNAP and OAuth 2.0 will likely exist in parallel for many deployments, and considerations have been taken to facilitate the mapping and transition from legacy systems to GNAP. Some examples of these can be found in Appendix D.5.

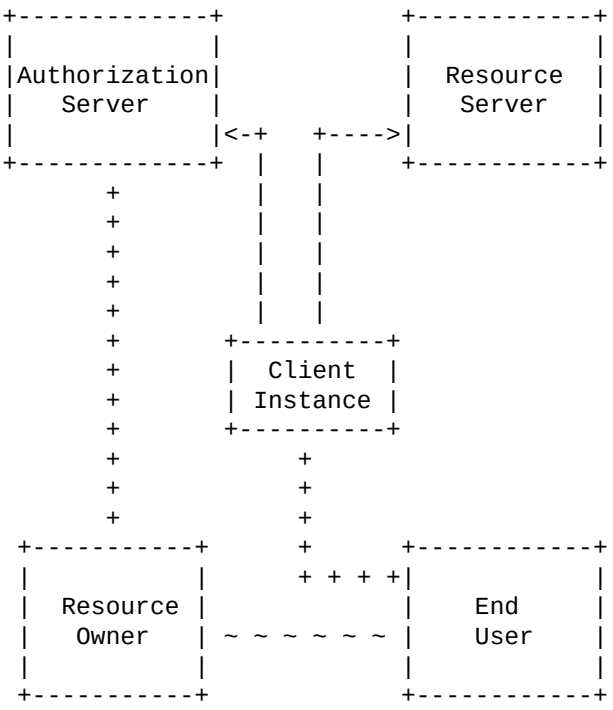
1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document contains non-normative examples of partial and complete HTTP messages, JSON structures, URLs, query components, keys, and other elements. Some examples use a single trailing backslash ’’ to indicate line wrapping for long values, as per [RFC8792]. The "\" character and leading spaces on wrapped lines are not part of the value.

1.2. Roles

The parties in GNAP perform actions under different roles. Roles are defined by the actions taken and the expectations leveraged on the role by the overall protocol.



Legend

+ + + indicates interaction between a human and computer
----- indicates interaction between two pieces of software
~ ~ ~ indicates a potential equivalence or out-of-band communication between roles

Authorization Server (AS) server that grants delegated privileges to a particular instance of client software in the form of access tokens or other information (such as subject information).

Client application operated by an end-user that consumes resources from one or several RSs, possibly requiring access privileges from one or several ASs.

Example: a client can be a mobile application, a web application, etc.

Note: this specification differentiates between a specific instance (the client instance, identified by its unique key) and the software running the instance (the client software). For some kinds of client software, there could be many instances of that software, each instance with a different key.

Resource Server (RS) server that provides operations on protected resources, where operations require a valid access token issued by an AS.

Resource Owner (RO) subject entity that may grant or deny operations on resources it has authority upon.

Note: the act of granting or denying an operation may be manual (i.e. through an interaction with a physical person) or automatic (i.e. through predefined organizational rules).

End-user natural person that operates a client instance.

Note: that natural person may or may not be the same entity as the RO.

The design of GNAP does not assume any one deployment architecture, but instead attempts to define roles that can be fulfilled in a number of different ways for different use cases. As long as a given role fulfills all of its obligations and behaviors as defined by the protocol, GNAP does not make additional requirements on its structure or setup.

Multiple roles can be fulfilled by the same party, and a given party can switch roles in different instances of the protocol. For example, the RO and end-user in many instances are the same person, where a user is authorizing the client instance to act on their own behalf at the RS. In this case, one party fulfills both of the RO and end-user roles, but the roles themselves are still defined separately from each other to allow for other use cases where they are fulfilled by different parties.

For another example, in some complex scenarios, an RS receiving requests from one client instance can act as a client instance for a downstream secondary RS in order to fulfill the original request. In this case, one piece of software is both an RS and a client instance from different perspectives, and it fulfills these roles separately as far as the overall protocol is concerned.

A single role need not be deployed as a monolithic service. For example, A client instance could have components that are installed on the end-user’s device as well as a back-end system that it communicates with. If both of these components participate in the delegation protocol, they are both considered part of the client instance. If there are several copies of the client software that run separately but all share the same key material, such as a deployed cluster, then this cluster is considered a single client instance.

In these cases, the distinct components of what is considered a GNAP client instance may use any number of different communication mechanisms between them, all of which would be considered an implementation detail of the client instances and out of scope of GNAP.

For another example, an AS could likewise be built out of many constituent components in a distributed architecture. The component that the client instance calls directly could be different from the component that the RO interacts with to drive consent, since API calls and user interaction have different security considerations in many environments. Furthermore, the AS could need to collect identity claims about the RO from one system that deals with user attributes while generating access tokens at another system that deals with security rights. From the perspective of GNAP, all of these are pieces of the AS and together fulfill the role of the AS as defined by the protocol. These pieces may have their own internal communications mechanisms which are considered out of scope of GNAP.

1.3. Elements

In addition to the roles above, the protocol also involves several elements that are acted upon by the roles throughout the process.

Attribute characteristics related to a subject.

Access Token a data artifact representing a set of rights and/or attributes.

Note: an access token can be first issued to an client instance (requiring authorization by the RO) and subsequently rotated.

Grant (verb): to permit an instance of client software to receive some attributes at a specific time and valid for a specific duration and/or to exercise some set of delegated rights to access a protected resource (noun): the act of granting.

Privilege right or attribute associated with a subject.

Note: the RO defines and maintains the rights and attributes associated to the protected resource, and might temporarily delegate some set of those privileges to an end-user. This process is refered to as privilege delegation.

Protected Resource protected API (Application Programming Interface) served by an RS and that can be accessed by a client, if and only if a valid access token is provided.

Note: to avoid complex sentences, the specification document may simply refer to resource instead of protected resource.

Right ability given to a subject to perform a given operation on a resource under the control of an RS.

Subject person, organization or device.

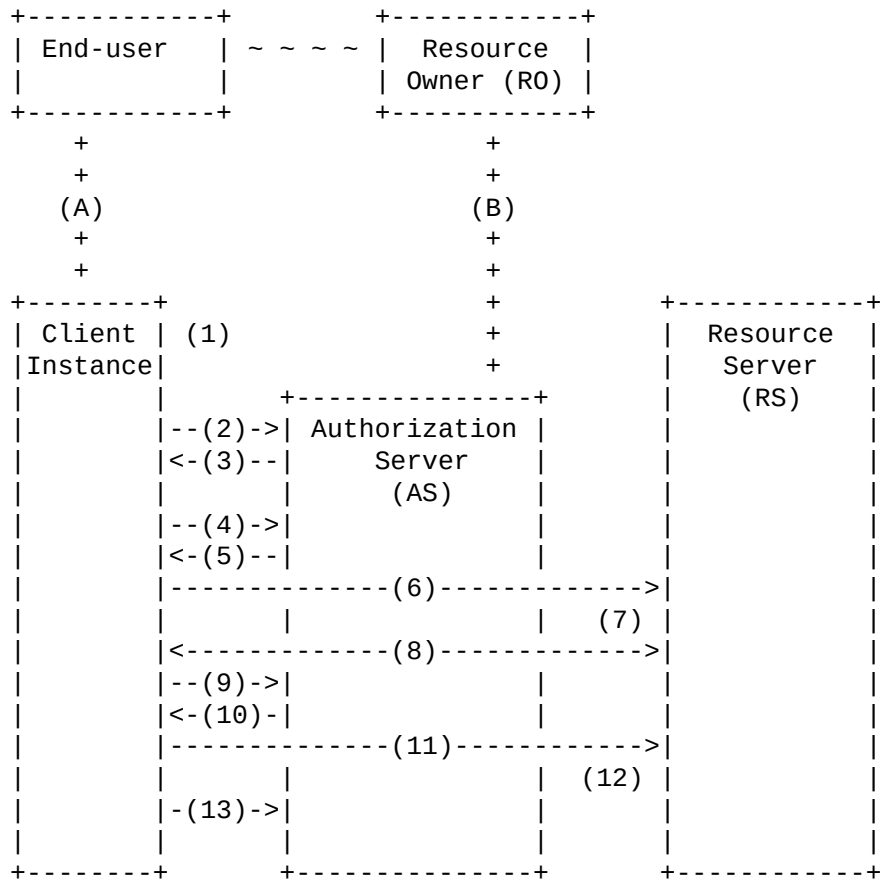
Subject Information statement asserted by an AS about a subject.

1.4. Sequences

GNAP can be used in a variety of ways to allow the core delegation process to take place. Many portions of this process are conditionally present depending on the context of the deployments, and not every step in this overview will happen in all circumstances.

Note that a connection between roles in this process does not necessarily indicate that a specific protocol message is sent across the wire between the components fulfilling the roles in question, or that a particular step is required every time. For example, for a client instance interested in only getting subject information directly, and not calling an RS, all steps involving the RS below do not apply.

In some circumstances, the information needed at a given stage is communicated out of band or is preconfigured between the components or entities performing the roles. For example, one entity can fulfil multiple roles, and so explicit communication between the roles is not necessary within the protocol flow. Additionally some components may not be involved in all use cases. For example, a client instance could be calling the AS just to get direct user information and have no need to get an access token to call an RS.



Legend
+ + + indicates a possible interaction with a human
----- indicates an interaction between protocol roles
~ ~ ~ indicates a potential equivalence or out-of-band communication between roles

* (A) The end-user interacts with the client instance to indicate a need for resources on behalf of the RO. This could identify the RS the client instance needs to call, the resources needed, or the RO that is needed to approve the request. Note that the RO and end-user are often the same entity in practice, but some more dynamic processes are discussed in [I-D.draft-ietf-gnap-resource-servers].

* (1) The client instance determines what access is needed and which AS to approach for access. Note that for most situations, the client instance is pre-configured with which AS to talk to and which kinds of access it needs.

* (2) The client instance requests access at the AS (Section 2).

* (3) The AS processes the request and determines what is needed to fulfill the request. The AS sends its response to the client instance (Section 3).

* (B) If interaction is required, the AS interacts with the RO (Section 4) to gather authorization. The interactive component of the AS can function using a variety of possible mechanisms including web page redirects, applications, challenge/response protocols, or other methods. The RO approves the request for the client instance being operated by the end-user. Note that the RO and end-user are often the same entity in practice.

* (4) The client instance continues the grant at the AS (Section 5).

* (5) If the AS determines that access can be granted, it returns a response to the client instance (Section 3) including an access token (Section 3.2) for calling the RS and any directly returned information (Section 3.4) about the RO.

* (6) The client instance uses the access token (Section 7.2) to call the RS.

* (7) The RS determines if the token is sufficient for the request by examining the token. The means of the RS determining this access are out of scope of this specification, but some options are discussed in [I-D.draft-ietf-gnap-resource-servers].

* (8) The client instance calls the RS (Section 7.2) using the access token until the RS or client instance determine that the token is no longer valid.

* (9) When the token no longer works, the client instance fetches an updated access token (Section 6.1) based on the rights granted in (5).

* (10) The AS issues a new access token (Section 3.2) to the client instance.

* (11) The client instance uses the new access token (Section 7.2) to call the RS.

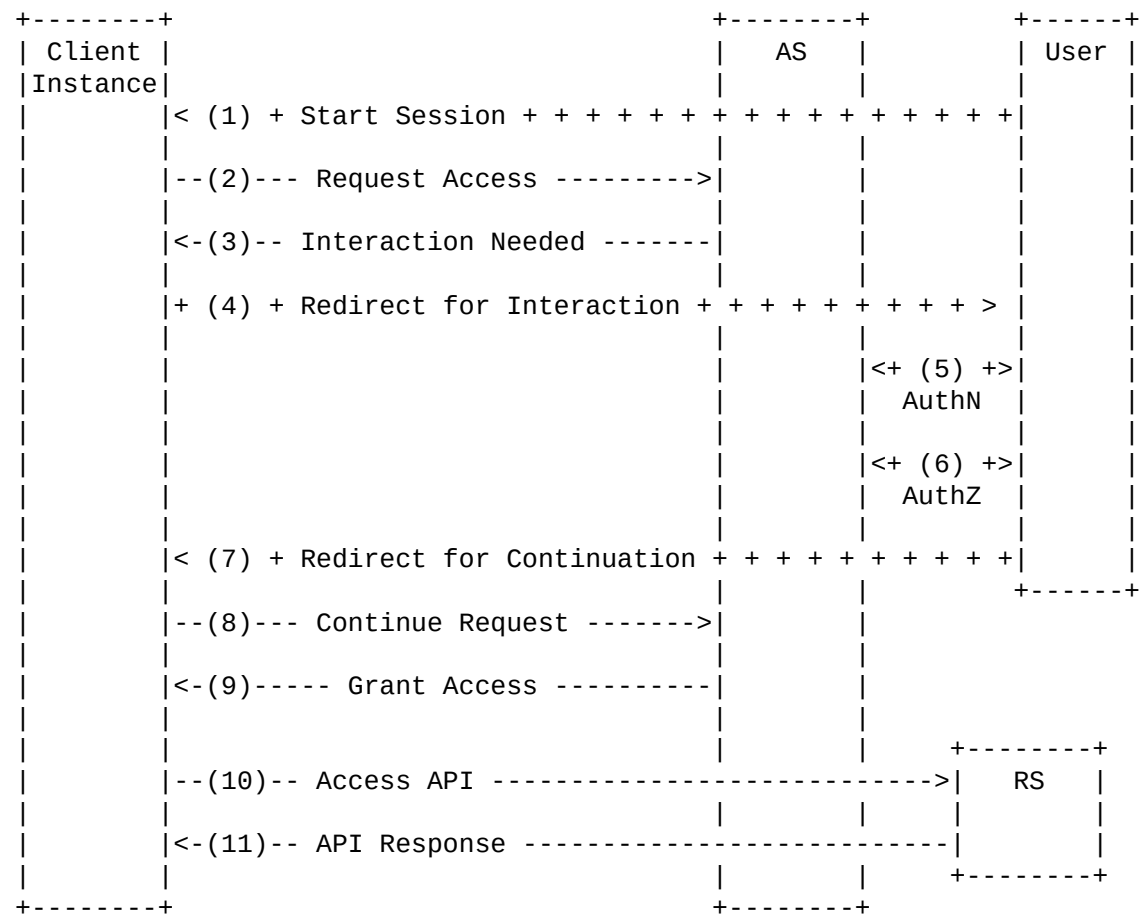
* (12) The RS determines if the new token is sufficient for the request. The means of the RS determining this access are out of scope of this specification, but some options are discussed in [I-D.draft-ietf-gnap-resource-servers].

* (13) The client instance disposes of the token (Section 6.2) once the client instance has completed its access of the RS and no longer needs the token.

The following sections and Appendix D contain specific guidance on how to use GNAP in different situations and deployments. For example, it is possible for the client instance to never request an access token and never call an RS, just as it is possible for there not to be a user involved in the delegation process.

1.4.1. Redirect-based Interaction

In this example flow, the client instance is a web application that wants access to resources on behalf of the current user, who acts as both the end-user and the resource owner (RO). Since the client instance is capable of directing the user to an arbitrary URL and receiving responses from the user's browser, interaction here is handled through front-channel redirects using the user's browser. The redirection URL used for interaction is a service hosted by the AS in this example. The client instance uses a persistent session with the user to ensure the same user that is starting the interaction is the user that returns from the interaction.



1. The client instance establishes a verifiable session to the user, in the role of the end-user.

2. The client instance requests access to the resource (Section 2). The client instance indicates that it can redirect to an arbitrary URL (Section 2.5.1.1) and receive a redirect from the browser (Section 2.5.2.1). The client instance stores verification information for its redirect in the session created in (1).

3. The AS determines that interaction is needed and responds (Section 3) with a URL to send the user to (Section 3.3.1) and information needed to verify the redirect (Section 3.3.4) in (7). The AS also includes information the client instance will need to continue the request (Section 3.1) in (8). The AS associates this continuation information with an ongoing request that will be referenced in (4), (6), and (8).

4. The client instance stores the verification and continuation information from (3) in the session from (1). The client instance then redirects the user to the URL (Section 4.1.1) given by the AS in (3). The user's browser loads the interaction redirect URL. The AS loads the pending request based on the incoming URL generated in (3).

5. The user authenticates at the AS, taking on the role of the RO.

6. As the RO, the user authorizes the pending request from the client instance.

7. When the AS is done interacting with the user, the AS redirects the user back (Section 4.2.1) to the client instance using the redirect URL provided in (2). The redirect URL is augmented with an interaction reference that the AS associates with the ongoing request created in (2) and referenced in (4). The redirect URL is also augmented with a hash of the security information provided in (2) and (3). The client instance loads the verification information from (2) and (3) from the session created in (1). The client instance calculates a hash (Section 4.2.3) based on this information and continues only if the hash validates. Note that the client instance needs to ensure that the parameters for the incoming request match those that it is expecting from the session created in (1). The client instance also needs to be prepared for the end-user never being returned to the client instance and handle timeouts appropriately.

8. The client instance loads the continuation information from (3) and sends the interaction reference from (7) in a request to continue the request (Section 5.1). The AS validates the interaction reference ensuring that the reference is associated with the request being continued.

9. If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the client instance.

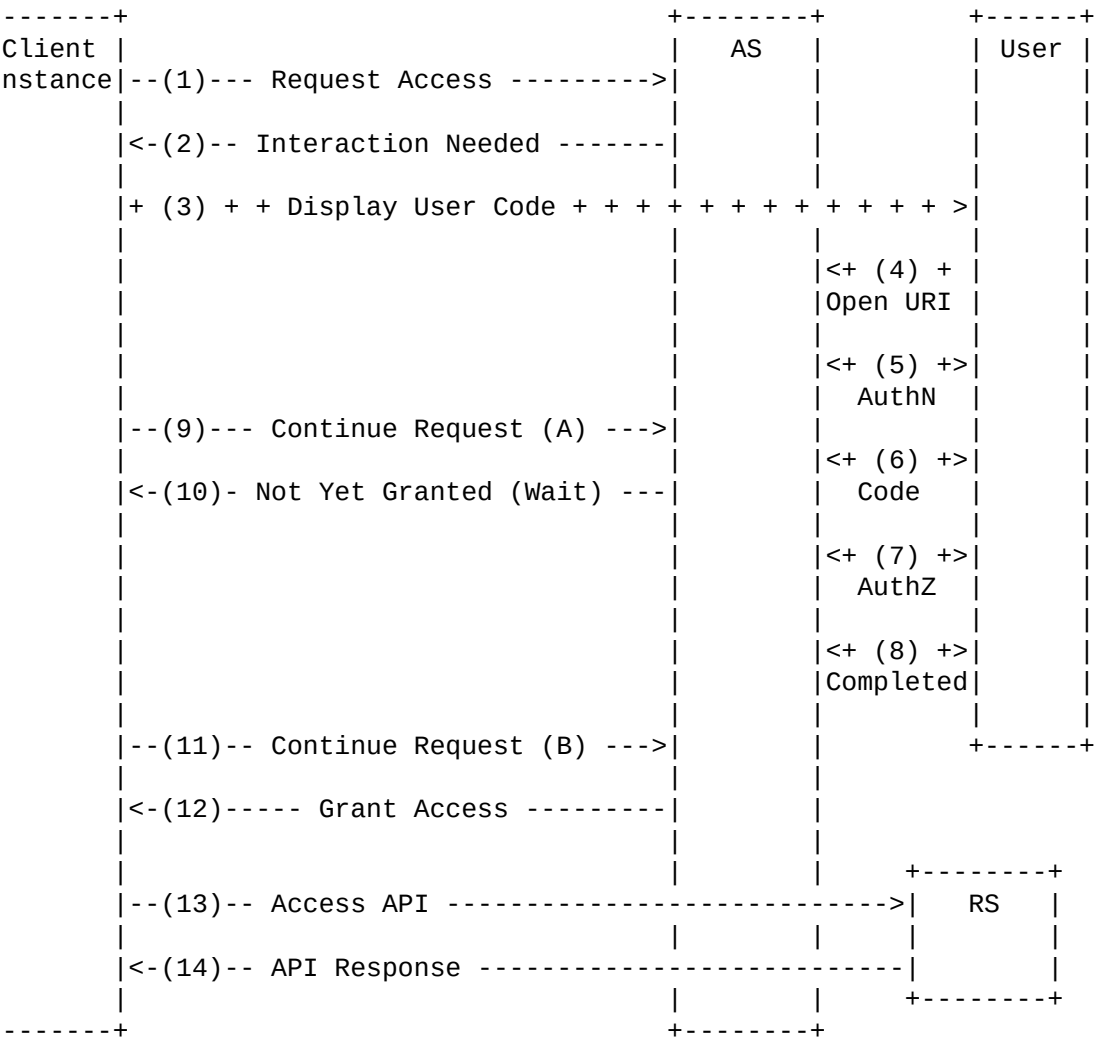
10. The client instance uses the access token (Section 7.2) to call the RS.

11. The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix D.1.

1.4.2. User-code Interaction

In this example flow, the client instance is a device that is capable of presenting a short, human-readable code to the user and directing the user to enter that code at a known URL. The URL the user enters the code at is an interactive service hosted by the AS in this example. The client instance is not capable of presenting an arbitrary URL to the user, nor is it capable of accepting incoming HTTP requests from the user's browser. The client instance polls the AS while it is waiting for the RO to authorize the request. The user's interaction is assumed to occur on a secondary device. In this example it is assumed that the user is both the end-user and RO, though the user is not assumed to be interacting with the client instance through the same web browser used for interaction at the AS.



1. The client instance requests access to the resource (Section 2). The client instance indicates that it can display a user code (Section 2.5.1.3).

2. The AS determines that interaction is needed and responds (Section 3) with a user code to communicate to the user (Section 3.3.3). This could optionally include a URL to direct the user to, but this URL should be static and so could be configured in the client instance's documentation. The AS also includes information the client instance will need to continue the request (Section 3.1) in (8) and (10). The AS associates this continuation information with an ongoing request that will be referenced in (4), (6), (8), and (10).

3. The client instance stores the continuation information from (2) for use in (8) and (10). The client instance then communicates the code to the user (Section 4.1.1) given by the AS in (2).

4. The user's directs their browser to the user code URL. This URL is stable and can be communicated via the client software's documentation, the AS documentation, or the client software itself. Since it is assumed that the RO will interact with the AS through a secondary device, the client instance does not provide a mechanism to launch the RO's browser at this URL.

5. The end-user authenticates at the AS, taking on the role of the RO.

6. The RO enters the code communicated in (3) to the AS. The AS validates this code against a current request in process.

7. As the RO, the user authorizes the pending request from the client instance.

8. When the AS is done interacting with the user, the AS indicates to the RO that the request has been completed.

9. Meanwhile, the client instance loads the continuation information stored at (3) and continues the request (Section 5). The AS determines which ongoing access request is referenced here and checks its state.

10. If the access request has not yet been authorized by the RO in (6), the AS responds to the client instance to continue the request (Section 3.1) at a future time through additional polled continuation requests. This response can include updated continuation information as well as information regarding how long the client instance should wait before calling again. The

client instance replaces its stored continuation information from the previous response (2). Note that the AS may need to determine that the RO has not approved the request in a sufficient amount of time and return an appropriate error to the client instance.

11. The client instance continues to poll the AS (Section 5.2) with the new continuation information in (9).

12. If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the client instance.

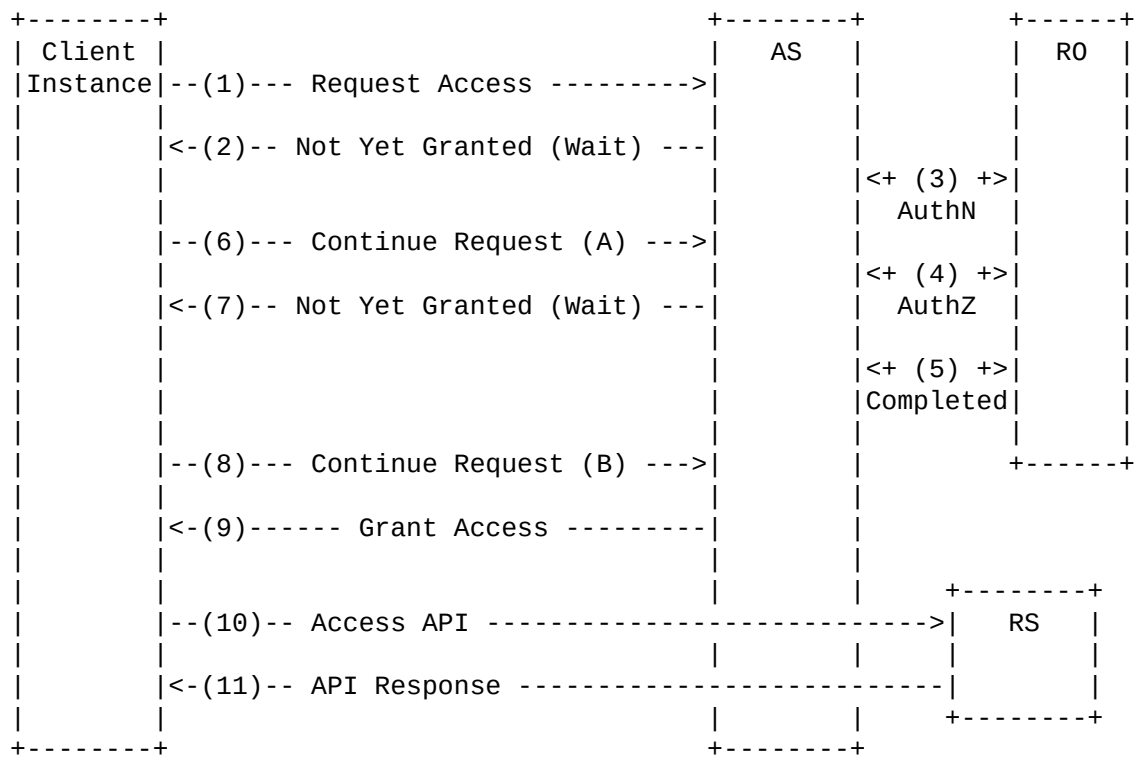
13. The client instance uses the access token (Section 7.2) to call the RS.

14. The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix D.2.

1.4.3. Asynchronous Authorization

In this example flow, the end-user and RO roles are fulfilled by different parties, and the RO does not interact with the client instance. The AS reaches out asynchronously to the RO during the request process to gather the RO's authorization for the client instance's request. The client instance polls the AS while it is waiting for the RO to authorize the request.



1. The client instance requests access to the resource (Section 2). The client instance does not send any interactions modes to the server, indicating that it does not expect to interact with the RO. The client instance can also signal which RO it requires authorization from, if known, by using the user request section (Section 2.4).

2. The AS determines that interaction is needed, but the client instance cannot interact with the RO. The AS responds (Section 3) with the information the client instance will need to continue the request (Section 3.1) in (6) and (8), including a signal that the client instance should wait before checking the status of the request again. The AS associates this continuation information with an ongoing request that will be referenced in (3), (4), (5), (6), and (8).

3. The AS determines which RO to contact based on the request in (1), through a combination of the user request (Section 2.4), the resources request (Section 2.1), and other policy information. The AS contacts the RO and authenticates them.

4. The RO authorizes the pending request from the client instance.

5. When the AS is done interacting with the RO, the AS indicates to the RO that the request has been completed.

6. Meanwhile, the client instance loads the continuation information stored at (2) and continues the request (Section 5). The AS determines which ongoing access request is referenced here and checks its state.

7. If the access request has not yet been authorized by the RO in (6), the AS responds to the client instance to continue the request (Section 3.1) at a future time through additional polling. This response can include refreshed credentials as well as information regarding how long the client instance should wait before calling again. The client instance replaces its stored continuation information from the previous response (2). Note that the AS may need to determine that the RO has not approved the request in a sufficient amount of time and return an appropriate error to the client instance.

8. The client instance continues to poll the AS (Section 5.2) with the new continuation information from (7).

9. If the request has been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) and direct subject information (Section 3.4) to the client instance.

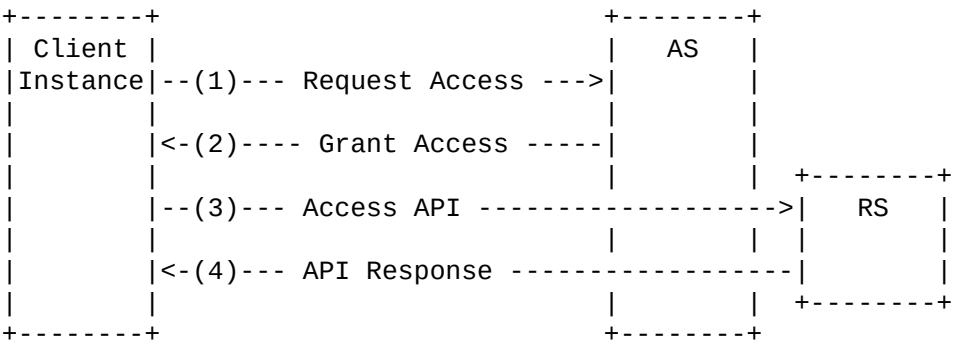
10. The client instance uses the access token (Section 7.2) to call the RS.

11. The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix D.4.

1.4.4. Software-only Authorization

In this example flow, the AS policy allows the client instance to make a call on its own behalf, without the need for a RO to be involved at runtime to approve the decision. Since there is no explicit RO, the client instance does not interact with an RO.



1. The client instance requests access to the resource (Section 2). The client instance does not send any interactions modes to the server.

2. The AS determines that the request is been authorized, the AS grants access to the information in the form of access tokens (Section 3.2) to the client instance. Note that direct subject information (Section 3.4) is not generally applicable in this use case, as there is no user involved.

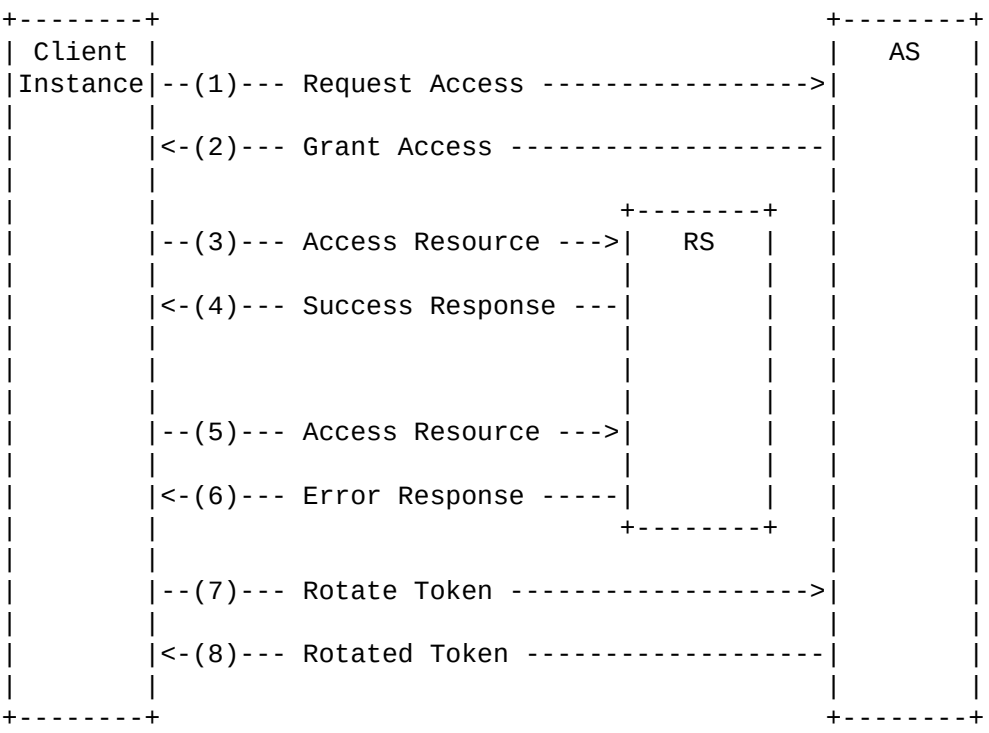
3. The client instance uses the access token (Section 7.2) to call the RS.

4. The RS validates the access token and returns an appropriate response for the API.

An example set of protocol messages for this method can be found in Appendix D.3.

1.4.5. Refreshing an Expired Access Token

In this example flow, the client instance receives an access token to access a resource server through some valid GNAP process. The client instance uses that token at the RS for some time, but eventually the access token expires. The client instance then gets a new access token by rotating the expired access token at the AS using the token's management URL.



1. The client instance requests access to the resource (Section 2).

2. The AS grants access to the resource (Section 3) with an access token (Section 3.2) usable at the RS. The access token response includes a token management URI.

3. The client instance uses the access token (Section 7.2) to call the RS.

4. The RS validates the access token and returns an appropriate response for the API.

5. Time passes and the client instance uses the access token to call the RS again.

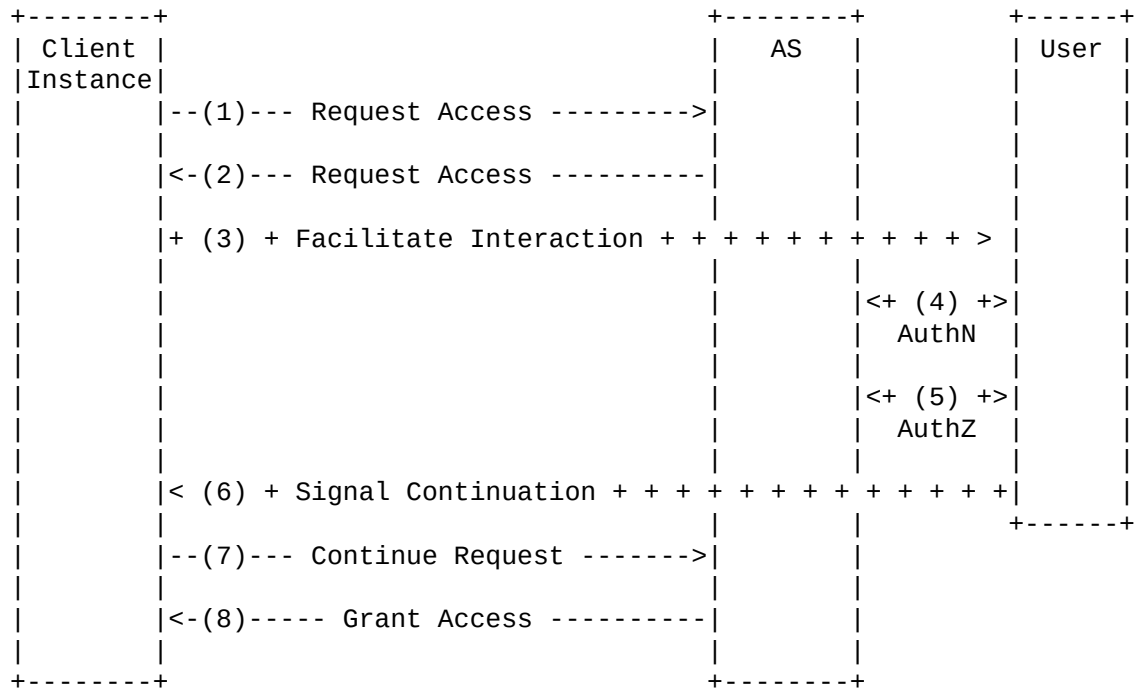
6. The RS validates the access token and determines that the access token is expired The RS responds to the client instance with an error.

7. The client instance calls the token management URI returned in (2) to rotate the access token (Section 6.1). The client instance uses the access token (Section 7.2) in this call as well as the appropriate key, see the token rotation section for details.

8. The AS validates the rotation request including the signature and keys presented in (5) and returns a new access token (Section 3.2.1). The response includes a new access token and can also include updated token management information, which the client instance will store in place of the values returned in (2).

1.4.6. Requesting User Information

In this scenario, the client instance does not call an RS and does not request an access token. Instead, the client instance only requests and is returned direct subject information (Section 3.4). Many different interaction modes can be used in this scenario, so these are shown only in the abstract as functions of the AS here.



1. The client instance requests access to subject information (Section 2).

2. The AS determines that interaction is needed and responds (Section 3) with appropriate information for facilitating user interaction (Section 3.3).

3. The client instance facilitates the user interacting with the AS (Section 4) as directed in (2).

4. The user authenticates at the AS, taking on the role of the RO.

5. As the RO, the user authorizes the pending request from the client instance.

6. When the AS is done interacting with the user, the AS returns the user to the client instance and signals continuation.

7. The client instance loads the continuation information from (2) and calls the AS to continue the request (Section 5).

8. If the request has been authorized, the AS grants access to the requested direct subject information (Section 3.4) to the client instance. At this stage, the user is generally considered "logged in" to the client instance based on the identifiers and assertions provided by the AS. Note that the AS can restrict the subject information returned and it might not match what the client instance requested, see the section on subject information for details.

2. Requesting Access

To start a request, the client instance sends JSON [RFC8259] document with an object as its root. Each member of the request object represents a different aspect of the client instance's request. Each field is described in detail in a section below.

access_token (object / array of objects) Describes the rights and properties associated with the requested access token. Section 2.1

subject (object) Describes the information about the RO that the client instance is requesting to be returned directly in the response from the AS. Section 2.2

client (object / string) Describes the client instance that is making this request, including the key that the client instance will use to protect this request and any continuation requests at the AS and any user-facing information about the client instance used in interactions. Section 2.3

user (object / string) Identifies the end-user to the AS in a manner that the AS can verify, either directly or by interacting with the end-user to determine their status as the RO. Section 2.4

interact (object) Describes the modes that the client instance has

for allowing the RO to interact with the AS and modes for the client instance to receive updates when interaction is complete. Section 2.5

Additional members of this request object can be defined by extensions to this protocol as described in Section 2.6

A non-normative example of a grant request is below:

```
{
    "access_token": {
        "access": [
            {
                "type": "photo-api",
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            },
            "dolphin-metadata"
        ]
    },
    "client": {
      "display": {
        "name": "My Client Display Name",
        "uri": "https://example.net/client"
      },
      "key": {
        "proof": "httpsig",
        "jwk": {
                    "kty": "RSA",
                    "e": "AQAB",
                    "kid": "xyz-1",
                    "alg": "RS256",
                    "n": "kOB5rR4Jv0GMeL...."
        }
      }
    },
    "interact": {
```

```
            "start": ["redirect"],
            "finish": {
                "method": "redirect",
                "uri": "https://client.example.net/return/123455",
                "nonce": "LKLTI25DK82FX4T4QFZC"
            }
        },
        "subject": {
            "formats": ["iss_sub", "opaque"],
            "assertions": ["id_token"]
        }
    }
```

The request and response MUST be sent as a JSON object in the body of the HTTP POST request with Content-Type "application/json", unless otherwise specified by the signature mechanism.

The authorization server MUST include the HTTP "Cache-Control" response header field [RFC7234] with a value set to "no-store".

2.1. Requesting Access to Resources

If the client instance is requesting one or more access tokens for the purpose of accessing an API, the client instance MUST include an "access_token" field. This field MUST be an object (for a single access token (Section 2.1.1)) or an array of these objects (for multiple access tokens (Section 2.1.2)), as described in the following sections.

2.1.1. Requesting a Single Access Token

To request a single access token, the client instance sends an "acccess_token" object composed of the following fields.

access (array of objects/strings) Describes the rights that the client instance is requesting for one or more access tokens to be used at RS's. This field is REQUIRED. Section 8

label (string) A unique name chosen by the client instance to refer to the resulting access token. The value of this field is opaque to the AS. If this field is included in the request, the AS MUST include the same label in the token response (Section 3.2). This field is REQUIRED if used as part of a multiple access token request (Section 2.1.2), and is OPTIONAL otherwise.

flags (array of strings) A set of flags that indicate desired attributes or behavior to be attached to the access token by the AS. This field is OPTIONAL.

The values of the "flags" field defined by this specification are as follows:

"bearer" If this flag is included, the access token being requested is a bearer token. If this flag is omitted, the access token is bound to the key used by the client instance in this request, or the key's most recent rotation. Methods for presenting bound and bearer access tokens are described in Section 7.2. [[See issue #38 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/38)]]

"split" If this flag is included, the client instance is capable of receiving a different number of tokens than specified in the token request (Section 2.1), including receiving multiple access tokens (Section 3.2.2) in response to any single token request (Section 2.1.1) or a different number of access tokens than requested in a multiple access token request (Section 2.1.2). The "label" fields of the returned additional tokens are chosen by the AS. The client instance MUST be able to tell from the token response where and how it can use each of the access tokens. [[See issue #37 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/37)]]

Flag values MUST NOT be included more than once.

Additional flags can be defined by extensions using a registry TBD (Section 11).

In the following example, the client instance is requesting access to a complex resource described by a pair of access request object.

```
"access_token": {
    "access": [
        {
            "type": "photo-api",
            "actions": [
                "read",
                "write",
                "delete"
            ],
            "locations": [
                "https://server.example.net/",
                "https://resource.local/other"
            ],
            "datatypes": [
                "metadata",
                "images"
            ]
        },
        {
            "type": "walrus-access",
            "actions": [
                "foo",
                "bar"
            ],
            "locations": [
                "https://resource.other/"
            ],
            "datatypes": [
                "data",
                "pictures",
                "walrus whiskers"
            ]
        }
    ],
    "label": "token1-23",
    "flags": [ "split" ]
}
```

If access is approved, the resulting access token is valid for the described resource and is bound to the client instance's key (or its most recent rotation). The token is labeled "token1-23" and could be split into multiple access tokens by the AS, if the AS chooses. The token response structure is described in Section 3.2.1.

2.1.2. Requesting Multiple Access Tokens

To request multiple access tokens to be returned in a single response, the client instance sends an array of objects as the value of the "access_token" parameter. Each object MUST conform to the request format for a single access token request, as specified in requesting a single access token (Section 2.1.1). Additionally, each object in the array MUST include the "label" field, and all values of these fields MUST be unique within the request. If the client instance does not include a "label" value for any entry in the array, or the values of the "label" field are not unique within the array, the AS MUST return an error.

The following non-normative example shows a request for two separate access tokens, "token1" and "token2".

```
   "access_token": [
       {
           "label": "token1",
           "access": [
               {
                   "type": "photo-api",
                   "actions": [
                       "read",
                       "write",
                       "dolphin"
                   ],
                   "locations": [
                       "https://server.example.net/",
                       "https://resource.local/other"
                   ],
                   "datatypes": [
                       "metadata",
                       "images"
                   ]
               },
               "dolphin-metadata"
           ]
       },
       {
           "label": "token2",
           "access": [
               {
                   "type": "walrus-access",
                   "actions": [
                       "foo",
                       "bar"
                   ],
                   "locations": [
                       "https://resource.other/"
                   ],
                   "datatypes": [
                       "data",
                       "pictures",
                       "walrus whiskers"
                   ]
               }
           ],
           "flags": [ "bearer" ]
       }
   ]
```

All approved access requests are returned in the multiple access token response (Section 3.2.2) structure using the values of the "label" fields in the request.

2.2. Requesting Subject Information

If the client instance is requesting information about the RO from the AS, it sends a "subject" field as a JSON object. This object MAY contain the following fields (or additional fields defined in a registry TBD (Section 11)).

formats (array of strings) An array of subject identifier subject types requested for the RO, as defined by [I-D.ietf-secevent-subject-identifiers].

assertions (array of strings) An array of requested assertion formats. Possible values include "id_token" for an [OIDC] ID Token and "saml2" for a SAML 2 assertion. Additional assertion values are defined by a registry TBD (Section 11). [[See issue #41 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/41)]]

```
"subject": {
   "formats": [ "iss_sub", "opaque" ],
   "assertions": [ "id_token", "saml2" ]
}
```

The AS can determine the RO's identity and permission for releasing this information through interaction with the RO (Section 4), AS policies, or assertions presented by the client instance (Section 2.4). If this is determined positively, the AS MAY return the RO's information in its response (Section 3.4) as requested.

Subject identifier types requested by the client instance serve only to identify the RO in the context of the AS and can't be used as communication channels by the client instance, as discussed in Section 3.4.

The AS SHOULD NOT re-use subject identifiers for multiple different ROs.

Note: the "formats" and "assertions" request fields are independent of each other, and a returned assertion MAY omit a requested subject identifier.

[[See issue #43 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/43)]]

2.3. Identifying the Client Instance

When sending a non-continuation request to the AS, the client instance MUST identify itself by including the "client" field of the request and by signing the request as described in Section 7.3. Note that for a continuation request (Section 5), the client instance is identified by its association with the request being continued and so this field is not sent under those circumstances.

When client instance information is sent by value, the "client" field of the request consists of a JSON object with the following fields.

key (object / string) The public key of the client instance to be used in this request as described in Section 7.1 or a reference to a key as described in Section 7.1.1. This field is REQUIRED.

class_id (string) An identifier string that the AS can use to identify the client software comprising this client instance. The contents and format of this field are up to the AS. This field is OPTIONAL.

display (object) An object containing additional information that the AS MAY display to the RO during interaction, authorization, and management. This field is OPTIONAL.

```
"client": {
    "key": {
        "proof": "httpsig",
        "jwk": {
                    "kty": "RSA",
                    "e": "AQAB",
                    "kid": "xyz-1",
                    "alg": "RS256",
                    "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8..."
        },
        "cert": "MIIEHDCCAwSgAwIBAgIBATANBgkqhkiG9w0BAQsFA..."
    },
    "class_id": "web-server-1234",
    "display": {
        "name": "My Client Display Name",
        "uri": "https://example.net/client"
    }
}
```

Additional fields are defined in a registry TBD (Section 11).

The client instance MUST prove possession of any presented key by the "proof" mechanism associated with the key in the request. Proof types are defined in a registry TBD (Section 11) and an initial set of methods is described in Section 7.3.

Note that the AS MAY know the client instance's public key ahead of time, and the AS MAY apply different policies to the request depending on what has been registered against that key. If the same public key is sent by value on subsequent access requests, the AS SHOULD treat these requests as coming from the same client instance for purposes of identification, authentication, and policy application. If the AS does not know the client instance's public key ahead of time, the AS MAY accept or reject the request based on AS policy, attestations within the "client" request, and other mechanisms.

[[See issue #44 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/44)]]

2.3.1. Identifying the Client Instance by Reference

If the client instance has an instance identifier that the AS can use to determine appropriate key information, the client instance can send this instance identifier as a direct reference value in lieu of the "client" object. The instance identifier MAY be assigned to a client instance at runtime through the Section 3.5 or MAY be obtained in another fashion, such as a static registration process at the AS.

"client": "client-541-ab"

When the AS receives a request with an instance identifier, the AS MUST ensure that the key used to sign the request (Section 7.3) is associated with the instance identifier.

If the AS does not recognize the instance identifier, the request MUST be rejected with an error.

If the client instance is identified in this manner, the registered key for the client instance MAY be a symmetric key known to the AS. The client instance MUST NOT send a symmetric key by value in the request, as doing so would expose the key directly instead of proving possession of it.

2.3.2. Providing Displayable Client Instance Information

If the client instance has additional information to display to the RO during any interactions at the AS, it MAY send that information in the "display" field. This field is a JSON object that declares information to present to the RO during any interactive sequences.

name (string) Display name of the client software

uri (string) User-facing web page of the client software

logo_uri (string) Display image to represent the client software

```
"display": {
    "name": "My Client Display Name",
    "uri": "https://example.net/client"
}
```

[[See issue #48 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/48)]]

Additional display fields are defined by a registry TBD (Section 11).

The AS SHOULD use these values during interaction with the RO. The values are for informational purposes only and MUST NOT be taken as authentic proof of the client instance's identity or source. The AS MAY restrict display values to specific client instances, as identified by their keys in Section 2.3.

2.3.3. Authenticating the Client Instance

If the presented key is known to the AS and is associated with a single instance of the client software, the process of presenting a key and proving possession of that key is sufficient to authenticate the client instance to the AS. The AS MAY associate policies with the client instance identified by this key, such as limiting which resources can be requested and which interaction methods can be used. For example, only specific client instances with certain known keys might be trusted with access tokens without the AS interacting directly with the RO as in Appendix D.3.

The presentation of a key allows the AS to strongly associate multiple successive requests from the same client instance with each other. This is true when the AS knows the key ahead of time and can use the key to authenticate the client instance, but also if the key is ephemeral and created just for this series of requests. As such the AS MAY allow for client instances to make requests with unknown keys. This pattern allows for ephemeral client instances, such as

single-page applications, and client software with many individual long-lived instances, such as mobile applications, to generate key pairs per instance and use the keys within the protocol without having to go through a separate registration step. The AS MAY limit which capabilities are made available to client instances with unknown keys. For example, the AS could have a policy saying that only previously-registered client instances can request particular resources, or that all client instances with unknown keys have to be interactively approved by an RO.

2.4. Identifying the User

If the client instance knows the identity of the end-user through one or more identifiers or assertions, the client instance MAY send that information to the AS in the "user" field. The client instance MAY pass this information by value or by reference.

sub_ids (array of objects) An array of subject identifiers for the end-user, as defined by [I-D.ietf-secevent-subject-identifiers].

assertions (object) An object containing assertions as values keyed on the assertion type defined by a registry TBD (Section 11). Possible keys include "id_token" for an [OIDC] ID Token and "saml2" for a SAML 2 assertion. Additional assertion values are defined by a registry TBD (Section 11). [[See issue #41 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/41)]]

```
"user": {
   "sub_ids": [ {
     "format": "opaque",
     "id": "J2G8G8O4AZ"
   } ],
   "assertions": {
     "id_token": "eyj..."
   }
}
```

Subject identifiers are hints to the AS in determining the RO and MUST NOT be taken as declarative statements that a particular RO is present at the client instance and acting as the end-user. Assertions SHOULD be validated by the AS. [[See issue #49 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/49)]]

If the identified end-user does not match the RO present at the AS during an interaction step, the AS SHOULD reject the request with an error.

[[See issue #50 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/50)]]

If the AS trusts the client instance to present verifiable assertions, the AS MAY decide, based on its policy, to skip interaction with the RO, even if the client instance provides one or more interaction modes in its request.

2.4.1. Identifying the User by Reference

User reference identifiers can be dynamically issued by the AS (Section 3.5) to allow the client instance to represent the same end-user to the AS over subsequent requests.

If the client instance has a reference for the end-user at this AS, the client instance MAY pass that reference as a string. The format of this string is opaque to the client instance.

"user": "XUT2MFM1XBIKJKSDU8QM"

User reference identifiers are not intended to be human-readable user identifiers or structured assertions. For the client instance to send either of these, use the full user request object (Section 2.4) instead.

[[See issue #51 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/51)]]

If the AS does not recognize the user reference, it MUST return an error.

2.5. Interacting with the User

Often, the AS will require interaction with the RO (Section 4) in order to approve a requested delegation to the client instance for both access to resources and direct subject information. Many times the end-user using the client instance is the same person as the RO, and the client instance can directly drive interaction with the end user by facilitating the process through means such as redirection to a URL or launching an application. Other times, the client instance can provide information to start the RO's interaction on a secondary device, or the client instance will wait for the RO to approve the request asynchronously. The client instance could also be signaled that interaction has concluded through a callback mechanism.

The client instance declares the parameters for interaction methods that it can support using the "interact" field.

The "interact" field is a JSON object with three keys whose values declare how the client can initiate and complete the request, as well as provide hints to the AS about user preferences such as locale. A client instance MUST NOT declare an interaction mode it does not support. The client instance MAY send multiple modes in the same request. There is no preference order specified in this request. An AS MAY respond to any, all, or none of the presented interaction modes (Section 3.3) in a request, depending on its capabilities and what is allowed to fulfill the request.

start (list of strings/objects) Indicates how the client instance can start an interaction.

finish (object) Indicates how the client instance can receive an indication that interaction has finished at the AS.

hints (object) Provides additional information to inform the interaction process at the AS.

The "interact" field MUST contain the "start" key, and MAY contain the "finish" and "hints" keys. The value of each key is an array which contains strings or JSON objects as defined below.

In this non-normative example, the client instance is indicating that it can redirect (Section 2.5.1.1) the end-user to an arbitrary URL and can receive a redirect (Section 2.5.2.1) through a browser request.

```
"interact": {
    "start": ["redirect"],
    "finish": {
        "method": "redirect",
        "uri": "https://client.example.net/return/123455",
        "nonce": "LKLTI25DK82FX4T4QFZC"
    }
}
```

In this non-normative example, the client instance is indicating that it can display a user code (Section 2.5.1.3) and direct the end-user to an arbitrary URL (Section 2.5.1.1) on a secondary device, but it cannot accept a redirect or push callback.

```
"interact": {
    "start": ["redirect", "user_code"]
}
```

If the client instance does not provide a suitable interaction mechanism, the AS cannot contact the RO asynchronously, and the AS determines that interaction is required, then the AS SHOULD return an error since the client instance will be unable to complete the request without authorization.

The AS SHOULD apply suitable timeouts to any interaction mechanisms provided, including user codes and redirection URLs. The client instance SHOULD apply suitable timeouts to any callback URLs.

2.5.1. Start Mode Definitions

This specification defines the following interaction start modes as an array of string values under the "start" key:

"redirect" Indicates that the client instance can direct the end-user to an arbitrary URL for interaction. Section 2.5.1.1

"app" Indicates that the client instance can launch an application on the end-user's device for interaction. Section 2.5.1.2

"user_code" Indicates that the client instance can communicate a human-readable short code to the end-user for use with a stable URL. Section 2.5.1.3

2.5.1.1. Redirect to an Arbitrary URL

If the client instance is capable of directing the end-user to a URL defined by the AS at runtime, the client instance indicates this by sending the "redirect" field with the boolean value "true". The means by which the client instance will activate this URL is out of scope of this specification, but common methods include an HTTP redirect, launching a browser on the end-user's device, providing a scannable image encoding, and printing out a URL to an interactive console. While this URL is generally hosted at the AS, the client instance can make no assumptions about its contents, composition, or relationship to the AS grant URL.

```
"interact": {
  "start": ["redirect"]
}
```

If this interaction mode is supported for this client instance and request, the AS returns a redirect interaction response Section 3.3.1. The client instance manages this interaction method as described in Section 4.1.1.

2.5.1.2. Open an Application-specific URL

If the client instance can open a URL associated with an application on the end-user's device, the client instance indicates this by sending the "app" field with boolean value "true". The means by which the client instance determines the application to open with this URL are out of scope of this specification.

```
"interact": {
   "start": ["app"]
}
```

If this interaction mode is supported for this client instance and request, the AS returns an app interaction response with an app URL payload Section 3.3.2. The client instance manages this interaction method as described in Section 4.1.3.

[[See issue #54 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/54)]]

2.5.1.3. Display a Short User Code

If the client instance is capable of displaying or otherwise communicating a short, human-entered code to the RO, the client instance indicates this by sending the "user_code" field with the boolean value "true". This code is to be entered at a static URL that does not change at runtime. While this URL is generally hosted at the AS, the client instance can make no assumptions about its contents, composition, or relationship to the AS grant URL.

```
"interact": {
    "start": ["user_code"]
}
```

If this interaction mode is supported for this client instance and request, the AS returns a user code and interaction URL as specified in Section 3.3.3. The client instances manages this interaction method as described in Section 4.1.2

2.5.2. Finish Interaction Modes

If the client instance is capable of receiving a message from the AS indicating that the RO has completed their interaction, the client instance indicates this by sending the following members of an object under the "finish" key.

method (string) REQUIRED. The callback method that the AS will use

to contact the client instance. This specification defines the following interaction completion methods, with other values defined by a registry TBD (Section 11):

"redirect" Indicates that the client instance can receive a redirect from the end-user's device after interaction with the RO has concluded. Section 2.5.2.1

"push" Indicates that the client instance can receive an HTTP POST request from the AS after interaction with the RO has concluded. Section 2.5.2.2

uri (string) REQUIRED. Indicates the URI that the AS will either send the RO to after interaction or send an HTTP POST request. This URI MAY be unique per request and MUST be hosted by or accessible by the client instance. This URI MUST NOT contain any fragment component. This URI MUST be protected by HTTPS, be hosted on a server local to the RO's browser ("localhost"), or use an application-specific URI scheme. If the client instance needs any state information to tie to the front channel interaction response, it MUST use a unique callback URI to link to that ongoing state. The allowable URIs and URI patterns MAY be restricted by the AS based on the client instance's presented key information. The callback URI SHOULD be presented to the RO during the interaction phase before redirect.

nonce (string) REQUIRED. Unique value to be used in the calculation of the "hash" query parameter sent to the callback URL, must be sufficiently random to be unguessable by an attacker. MUST be generated by the client instance as a unique value for this request.

hash_method (string) OPTIONAL. The hash calculation mechanism to be used for the callback hash in Section 4.2.3. Can be one of "sha3" or "sha2". If absent, the default value is "sha3". [[See issue #56 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/56)]]

If this interaction mode is supported for this client instance and request, the AS returns a nonce for use in validating the callback response (Section 3.3.4). Requests to the callback URI MUST be processed as described in Section 4.2, and the AS MUST require presentation of an interaction callback reference as described in Section 5.1.

[[See issue #58 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/58)]]

[[See issue #59 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/59)]]

2.5.2.1. Receive an HTTP Callback Through the Browser

A finish "method" value of "redirect" indicates that the client instance will expect a request from the RO's browser using the HTTP method GET as described in Section 4.2.1.

```
"interact": {
    "finish": {
        "method": "redirect",
        "uri": "https://client.example.net/return/123455",
        "nonce": "LKLTI25DK82FX4T4QFZC"
    }
}
```

Requests to the callback URI MUST be processed by the client instance as described in Section 4.2.1.

Since the incoming request to the callback URL is from the RO's browser, this method is usually used when the RO and end-user are the same entity. As such, the client instance MUST ensure the end-user is present on the request to prevent substitution attacks.

2.5.2.2. Receive an HTTP Direct Callback

A finish "method" value of "push" indicates that the client instance will expect a request from the AS directly using the HTTP method POST as described in Section 4.2.2.

```
"interact": {
    "finish": {
        "method": "push",
        "uri": "https://client.example.net/return/123455",
        "nonce": "LKLTI25DK82FX4T4QFZC"
    }
}
```

Requests to the callback URI MUST be processed by the client instance as described in Section 4.2.2.

Since the incoming request to the callback URL is from the AS and not from the RO's browser, the client instance MUST NOT require the end-user to be present on the incoming HTTP request.

[[See issue #60 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/60)]]

2.5.3. Hints

The "hints" key is an object describing one or more suggestions from the client instance that the AS can use to help drive user interaction.

This specification defines the following properties under the "hints" key:

ui_locales (array of strings) Indicates the end-user's preferred locales that the AS can use during interaction, particularly before the RO has authenticated. Section 2.5.3.1

The following sections detail requests for interaction modes. Additional interaction modes are defined in a registry TBD (Section 11).

2.5.3.1. Indicate Desired Interaction Locales

If the client instance knows the end-user's locale and language preferences, the client instance can send this information to the AS using the "ui_locales" field with an array of locale strings as defined by [RFC5646].

```
"interact": {
    "hints": {
        "ui_locales": ["en-US", "fr-CA"]
    }
}
```

If possible, the AS SHOULD use one of the locales in the array, with preference to the first item in the array supported by the AS. If none of the given locales are supported, the AS MAY use a default locale.

2.5.4. Extending Interaction Modes

Additional interaction start modes, finish modes, and hints are defined in a registry TBD (Section 11).

2.6. Extending The Grant Request

The request object MAY be extended by registering new items in a registry TBD (Section 11). Extensions SHOULD be orthogonal to other parameters. Extensions MUST document any aspects where the extension item affects or influences the values or behavior of other request and response objects.

3. Grant Response

In response to a client instance's request, the AS responds with a JSON object as the HTTP entity body. Each possible field is detailed in the sections below

continue (object) Indicates that the client instance can continue the request by making one or more continuation requests. Section 3.1

access_token (object / array of objects) A single access token or set of access tokens that the client instance can use to call the RS on behalf of the RO. Section 3.2.1

interact (object) Indicates that interaction through some set of defined mechanisms needs to take place. Section 3.3

subject (object) Claims about the RO as known and declared by the AS. Section 3.4

instance_id (string) An identifier this client instance can use to identify itself when making future requests. Section 3.5

user_handle (string) An identifier this client instance can use to identify its current end-user when making future requests. Section 3.5

error (object) An error code indicating that something has gone wrong. Section 3.6

In this example, the AS is returning an interaction URL (Section 3.3.1), a callback nonce (Section 3.3.4), and a continuation response (Section 3.1).

```
{
    "interact": {
        "redirect": "https://server.example.com/interact/4CF492ML\
          VMSW9MKMXKHQ",
        "finish": "MBDOFXG4Y5CVJCX821LH"
    },
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU",
        },
        "uri": "https://server.example.com/tx"
    }
}
```

In this example, the AS is returning a bearer access token (Section 3.2.1) with a management URL and a subject identifier (Section 3.4) in the form of an opaque identifier.

```
{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "flags": ["bearer"],
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
    },
    "subject": {
        "sub_ids": [ {
           "format": "opaque",
           "id": "J2G8G8O4AZ"
        } ]
    }
}
```

In this example, the AS is returning set of subject identifiers (Section 3.4), simultaneously as an opaque identifier, an email address, and a decentralized identifier (DID).

```
{
    "subject": {
        "sub_ids": [ {
           "subject_type": "opaque",
           "id": "J2G8G8O4AZ"
        }, {
           "format": "email",
           "email": "user@example.com"
        }, {
           "format": "did",
           "url": "did:example:123456"
        } ]
    }
}
```

3.1. Request Continuation

If the AS determines that the request can be continued with additional requests, it responds with the "continue" field. This field contains a JSON object with the following properties.

uri (string) REQUIRED. The URI at which the client instance can

make continuation requests. This URI MAY vary per request, or MAY be stable at the AS if the AS includes an access token. The client instance MUST use this value exactly as given when making a continuation request (Section 5).

wait (integer) RECOMMENDED. The amount of time in integer seconds the client instance SHOULD wait after receiving this continuation handle and calling the URI.

access_token (object) REQUIRED. A unique access token for continuing the request, in the format specified in Section 3.2.1. This access token MUST be bound to the client instance's key used in the request and MUST NOT be a "bearer" token. As a consequence, the "flags" array of this access token MUST NOT contain the string "bearer" and the "key" field MUST be omitted. This access token MUST NOT be usable at resources outside of the AS. The client instance MUST present the access token in all requests to the continuation URI as described in Section 7.2. [[See issue #66 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/66)]]

```
{
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 60
    }
}
```

The client instance can use the values of this field to continue the request as described in Section 5. Note that the client instance MUST sign all continuation requests with its key as described in Section 7.3 and MUST present the access token in its continuation request.

This field SHOULD be returned when interaction is expected, to allow the client instance to follow up after interaction has been concluded.

3.2. Access Tokens

If the AS has successfully granted one or more access tokens to the client instance, the AS responds with the "access_token" field. This field contains either a single access token as described in Section 3.2.1 or an array of access tokens as described in Section 3.2.2.

The client instance uses any access tokens in this response to call the RS as described in Section 7.2.

3.2.1. Single Access Token

If the client instance has requested a single access token and the AS has granted that access token, the AS responds with the "access_token" field. The value of this field is an object with the following properties.

value (string) REQUIRED. The value of the access token as a string. The value is opaque to the client instance. The value SHOULD be limited to ASCII characters to facilitate transmission over HTTP headers within other protocols without requiring additional encoding.

label (string) REQUIRED for multiple access tokens, OPTIONAL for single access token. The value of the "label" the client instance provided in the associated token request (Section 2.1), if present. If the token has been split by the AS, the value of the "label" field is chosen by the AS and the "split" field is included and set to "true".

manage (string) OPTIONAL. The management URI for this access token. If provided, the client instance MAY manage its access token as described in Section 6. This management URI is a function of the AS and is separate from the RS the client instance is requesting access to. This URI MUST NOT include the access token value and SHOULD be different for each access token issued in a request.

access (array of objects/strings) RECOMMENDED. A description of the rights associated with this access token, as defined in Section 8. If included, this MUST reflect the rights associated with the issued access token. These rights MAY vary from what was requested by the client instance.

expires_in (integer) OPTIONAL. The number of seconds in which the access will expire. The client instance MUST NOT use the access token past this time. An RS MUST NOT accept an access token past this time. Note that the access token MAY be revoked by the AS or RS at any point prior to its expiration.

key (object / string) OPTIONAL. The key that the token is bound to, if different from the client instance's presented key. The key MUST be an object or string in a format described in Section 7.1. The client instance MUST be able to dereference or process the key information in order to be able to sign the request.

flags (array of strings) OPTIONAL. A set of flags that represent attributes or behaviors of the access token issued by the AS.

The values of the "flags" field defined by this specification are as follows:

"bearer" This flag indicates whether the token is bound to the client instance's key. If the "bearer" flag is present, the access token is a bearer token, and the "key" field in this response MUST be omitted. If the "bearer" flag is omitted and the "key" field in this response is omitted, the token is bound the key used by the client instance (Section 2.3) in its request for access. If the "bearer" flag is omitted, and the "key" field is present, the token is bound to the key and proofing mechanism indicated in the "key" field.

"durable" OPTIONAL. Flag indicating a hint of AS behavior on token rotation. If this flag is present, then the client instance can expect a previously-issued access token to continue to work after it has been rotated (Section 6.1) or the underlying grant request has been modified (Section 5.3), resulting in the issuance of new access tokens. If this flag is omitted, the client instance can anticipate a given access token will stop working after token rotation or grant request modification. Note that a token flagged as "durable" can still expire or be revoked through any normal means.

"split" OPTIONAL. Flag indicating that this token was generated by issuing multiple access tokens in response to one of the client instance's token request (Section 2.1) objects. This behavior MUST NOT be used unless the client instance has specifically requested it by use of the "split" flag.

Flag values MUST NOT be included more than once.

Additional flags can be defined by extensions using a registry TBD (Section 11).

The following non-normative example shows a single access token bound to the client instance's key used in the initial request, with a management URL, and that has access to three described resources (one using an object and two described by reference strings).

```
   "access_token": {
       "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
       "manage": "https://server.example.com/token/PRY5NM33O\
           M4TB8N6BW7OZB8CDFONP219RP1L",
       "access": [
           {
               "type": "photo-api",
               "actions": [
                   "read",
                   "write",
                   "dolphin"
               ],
               "locations": [
                   "https://server.example.net/",
                   "https://resource.local/other"
               ],
               "datatypes": [
                   "metadata",
                   "images"
               ]
           },
           "read", "dolphin-metadata"
       ]
   }
```

The following non-normative example shows a single bearer access token with access to two described resources.

```
   "access_token": {
       "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
       "flags": ["bearer"],
       "access": [
           "finance", "medical"
       ]
   }
```

If the client instance requested a single access token (Section 2.1.1), the AS MUST NOT respond with the multiple access token structure unless the client instance sends the "split" flag as described in Section 2.1.1.

If the AS has split the access token response, the response MUST include the "split" flag.

[[See issue #69 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/69)]]

3.2.2. Multiple Access Tokens

If the client instance has requested multiple access tokens and the AS has granted at least one of them, the AS responds with the "access_token" field. The value of this field is a JSON array, the members of which are distinct access tokens as described in Section 3.2.1. Each object MUST have a unique "label" field, corresponding to the token labels chosen by the client instance in the multiple access token request (Section 2.1.2).

In this non-normative example, two tokens are issued under the names "token1" and "token2", and only the first token has a management URL associated with it.

```
"access_token": [
    {
        "label": "token1",
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [ "finance" ]
    },
    {
        "label": "token2",
        "value": "UFGLO2FDAFG7VGZZPJ3IZEMN21EVU71FHCARP4J1",
        "access": [ "medical" ]
    }
}
```

Each access token corresponds to one of the objects in the "access_token" array of the client instance's request (Section 2.1.2).

The multiple access token response MUST be used when multiple access tokens are requested, even if only one access token is issued as a result of the request. The AS MAY refuse to issue one or more of the requested access tokens, for any reason. In such cases the refused token is omitted from the response and all of the other issued access tokens are included in the response the requested names appropriate names.

If the client instance requested multiple access tokens (Section 2.1.2), the AS MUST NOT respond with a single access token structure, even if only a single access token is granted. In such cases, the AS responds with a multiple access token structure containing one access token.

If the AS has split the access token response, the response MUST include the "split" flag in the "flags" array.

```
"access_token": [
    {
        "label": "split-1",
        "value": "8N6BW7OZB8CDFONP219-OS9M2PMHKUR64TBRP1LT0",
        "flags": ["split"],
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [ "fruits" ]
    },
    {
        "label": "split-2",
        "value": "FG7VGZZPJ3IZEMN21EVU71FHCAR-UFGLO2FDAP4J1",
        "flags": ["split"],
        "access": [ "vegetables" ]
    }
}
```

Each access token MAY be bound to different keys with different proofing mechanisms.

If token management (Section 6) is allowed, each access token SHOULD have different "manage" URIs.

[[See issue #70 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/70)]]

3.3. Interaction Modes

If the client instance has indicated a capability to interact with the RO in its request (Section 2.5), and the AS has determined that interaction is both supported and necessary, the AS responds to the client instance with any of the following values in the "interact" field of the response. There is no preference order for interaction modes in the response, and it is up to the client instance to determine which ones to use. All supported interaction methods are included in the same "interact" object.

redirect (string) Redirect to an arbitrary URL. Section 3.3.1

app (string) Launch of an application URL. Section 3.3.2

finish (string) A nonce used by the client instance to verify the callback after interaction is completed. Section 3.3.4

user_code (object) Display a short user code. Section 3.3.3

Additional interaction mode responses can be defined in a registry TBD (Section 11).

The AS MUST NOT respond with any interaction mode that the client instance did not indicate in its request. The AS MUST NOT respond with any interaction mode that the AS does not support. Since interaction responses include secret or unique information, the AS SHOULD respond to each interaction mode only once in an ongoing request, particularly if the client instance modifies its request (Section 5.3).

3.3.1. Redirection to an arbitrary URL

If the client instance indicates that it can redirect to an arbitrary URL (Section 2.5.1.1) and the AS supports this mode for the client instance's request, the AS responds with the "redirect" field, which is a string containing the URL to direct the end-user to. This URL MUST be unique for the request and MUST NOT contain any security-sensitive information such as user identifiers or access tokens.

```
"interact": {
    "redirect": "https://interact.example.com/4CF492MLVMSW9MKMXKHQ"
}
```

The URL returned is a function of the AS, but the URL itself MAY be completely distinct from the URL the client instance uses to request access (Section 2), allowing an AS to separate its user-interactive functionality from its back-end security functionality. If the AS does not directly host the functionality accessed through the given URL, then the means for the interaction functionality to communicate with the rest of the AS are out of scope for this specification.

[[See issue #72 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/72)]]

The client instance sends the end-user to the URL to interact with the AS. The client instance MUST NOT alter the URL in any way. The means for the client instance to send the end-user to this URL is out of scope of this specification, but common methods include an HTTP redirect, launching the system browser, displaying a scannable code, or printing out the URL in an interactive console. See details of the interaction in Section 4.1.1.

3.3.2. Launch of an application URL

If the client instance indicates that it can launch an application URL (Section 2.5.1.2) and the AS supports this mode for the client instance's request, the AS responds with the "app" field, which is a string containing the URL for the client instance to launch. This URL MUST be unique for the request and MUST NOT contain any security-sensitive information such as user identifiers or access tokens.

```
"interact": {
    "app": "https://app.example.com/launch?tx=4CF492MLV"
}
```

The means for the launched application to communicate with the AS are out of scope for this specification.

The client instance launches the URL as appropriate on its platform, and the means for the client instance to launch this URL is out of scope of this specification. The client instance MUST NOT alter the URL in any way. The client instance MAY attempt to detect if an installed application will service the URL being sent before attempting to launch the application URL. See details of the interaction in Section 4.1.3.

[[See issue #71 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/71)]]

3.3.3. Display of a Short User Code

If the client instance indicates that it can display a short user-typeable code (Section 2.5.1.3) and the AS supports this mode for the client instance's request, the AS responds with a "user_code" field. This field is an object that contains the following members.

code (string) REQUIRED. A unique short code that the user can type into an authorization server. This string MUST be case-insensitive, MUST consist of only easily typeable characters (such as letters or numbers). The time in which this code will be accepted SHOULD be short lived, such as several minutes. It is RECOMMENDED that this code be no more than eight characters in length.

url (string) RECOMMENDED. The interaction URL that the client instance will direct the RO to. This URL MUST be stable such that client instances can be statically configured with it.

```
"interact": {
    "user_code": {
        "code": "A1BC-3DFF",
        "url": "https://srv.ex/device"
    }
}
```

The client instance MUST communicate the "code" to the end-user in some fashion, such as displaying it on a screen or reading it out audibly.

The client instance SHOULD also communicate the URL if possible to facilitate user interaction, but since the URL should be stable, the client instance should be able to safely decide to not display this value. As this interaction mode is designed to facilitate interaction via a secondary device, it is not expected that the client instance redirect the end-user to the URL given here at runtime. Consequently, the URL needs to be stable enough that a client instance could be statically configured with it, perhaps referring the end-user to the URL via documentation instead of through an interactive means. If the client instance is capable of communicating an arbitrary URL to the end-user, such as through a scannable code, the client instance can use the "redirect" (Section 2.5.1.1) mode for this purpose instead of or in addition to the user code mode.

The URL returned is a function of the AS, but the URL itself MAY be completely distinct from the URL the client instance uses to request access (Section 2), allowing an AS to separate its user-interactive functionality from its back-end security functionality. If the AS does not directly host the functionality accessed through the given URL, then the means for the interaction functionality to communicate with the rest of the AS are out of scope for this specification.

See details of the interaction in Section 4.1.2.

[[See issue #72 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/72)]]

3.3.4. Interaction Finish

If the client instance indicates that it can receive a post-interaction redirect or push at a URL (Section 2.5.2) and the AS supports this mode for the client instance's request, the AS responds with a "finish" field containing a nonce that the client instance will use in validating the callback as defined in Section 4.2.

```
"interact": {
    "finish": "MBDOFXG4Y5CVJCX821LH"
}
```

When the interaction is completed, the interaction component MUST contact the client instance using either a redirect or launch of the RO's browser or through an HTTP POST to the client instance's callback URL using the method indicated in the interaction request (Section 2.5.2) as described in Section 4.2.

If the AS returns a nonce, the client instance MUST NOT continue a grant request before it receives the associated interaction reference on the callback URI. See details in Section 4.2.

3.3.5. Extending Interaction Mode Responses

Extensions to this specification can define new interaction mode responses in a registry TBD (Section 11). Extensions MUST document the corresponding interaction request.

3.4. Returning User Information

If information about the RO is requested and the AS grants the client instance access to that data, the AS returns the approved information in the "subject" response field. This field is an object with the following OPTIONAL properties.

sub_ids (array of objects) An array of subject identifiers for the RO, as defined by [I-D.ietf-secevent-subject-identifiers].

assertions (object) An object containing assertions as values keyed on the assertion type defined by a registry TBD (Section 11). [[See issue #41 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/41)]]

updated_at (string) Timestamp as an ISO8610 date string, indicating when the identified account was last updated. The client instance MAY use this value to determine if it needs to request updated profile information through an identity API. The definition of such an identity API is out of scope for this specification.

```
"subject": {
   "sub_ids": [ {
     "format": "opaque",
     "id": "J2G8G8O4AZ"
   } ],
   "assertions": {
     "id_token": "eyj..."
   }
}
```

The AS MUST return the "subject" field only in cases where the AS is sure that the RO and the end-user are the same party. This can be accomplished through some forms of interaction with the RO (Section 4).

Subject identifiers returned by the AS SHOULD uniquely identify the RO at the AS. Some forms of subject identifier are opaque to the client instance (such as the subject of an issuer and subject pair), while others forms (such as email address and phone number) are intended to allow the client instance to correlate the identifier with other account information at the client instance. The AS MUST ensure that the returned subject identifiers only apply to the authenticated end user. The client instance MUST NOT request or use any returned subject identifiers for communication purposes (see Section 2.2). That is, a subject identifier returned in the format of an email address or a phone number only identifies the RO to the AS and does not indicate that the AS has validated that the represented email address or phone number in the identifier is suitable for communication with the current user. To get such information, the client instance MUST use an identity protocol to request and receive additional identity claims. The details of an identity protocol and associated schema are outside the scope of this specification.

Extensions to this specification MAY define additional response properties in a registry TBD (Section 11).

3.5. Returning Dynamically-bound Reference Handles

Many parts of the client instance’s request can be passed as either a value or a reference. The use of a reference in place of a value allows for a client instance to optimize requests to the AS.

Some references, such as for the client instance's identity (Section 2.3.1) or the requested resources (Section 8.1), can be managed statically through an admin console or developer portal provided by the AS or RS. The developer of the client software can include these values in their code for a more efficient and compact request.

If desired, the AS MAY also generate and return some of these references dynamically to the client instance in its response to facilitate multiple interactions with the same software. The client instance SHOULD use these references in future requests in lieu of sending the associated data value. These handles are intended to be used on future requests.

Dynamically generated handles are string values that MUST be protected by the client instance as secrets. Handle values MUST be unguessable and MUST NOT contain any sensitive information. Handle values are opaque to the client instance.

All dynamically generated handles are returned as fields in the root JSON object of the response. This specification defines the following dynamic handle returns, additional handles can be defined in a registry TBD (Section 11).

instance_id (string) A string value used to represent the information in the "client" object that the client instance can use in a future request, as described in Section 2.3.1.

user_handle (string) A string value used to represent the current user. The client instance can use in a future request, as described in Section 2.4.1.

This non-normative example shows two handles along side an issued access token.

```
{
    "user_handle": "XUT2MFM1XBIKJKSDU8QM",
    "instance_id": "7C7C4AZ9KHRS6X63AJAO",
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0"
    }
}
```

[[See issue #77 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/77)]]

[[See issue #78 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/78)]]

3.6. Error Response

If the AS determines that the request cannot be issued for any reason, it responds to the client instance with an error message.

error (string) The error code.

```
{

  "error": "user_denied"

}
```

The error code is one of the following, with additional values available in a registry TBD (Section 11):

user_denied The RO denied the request.

too_fast The client instance did not respect the timeout in the wait response.

unknown_request The request referenced an unknown ongoing access request.

[[See issue #79 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/79)]]

3.7. Extending the Response

Extensions to this specification MAY define additional fields for the grant response in a registry TBD (Section 11).

4. Determining Authorization and Consent

When the client instance makes its Section 2 to the AS for delegated access, it is capable of asking for several different kinds of information in response:

* the access being requested in the "access_token" request parameter

* the subject information being requested in the "subject" request parameter

* any additional requested information defined by extensions of this protocol

The AS determines what authorizations and consents are required to fulfill this requested delegation. The details of how the AS makes this determination are out of scope for this document. However, there are several common patterns defined and supported by GNAP for fulfilling these requirements, including information sent by the client instance, information gathered through the interaction process, and information supplied by external parties. An individual AS can define its own policies and processes for deciding when and how to gather the necessary authorizations and consent.

The client instance can supply information directly to the AS in its request. From this information, the AS can determine if the requested delegation can be granted immediately. The client instance can send several kinds of things, including:

* the identity of the client instance, known from the presented keys or associated identifiers

* the identity of the end user presented in the "user" request parameter

* any additional information presented by the client instance in the request, including any extensions

The AS will verify this presented information in the context of the client instance's request and can only trust the information as much as it trusts the presentation and context of the information. If the AS determines that the information presented in the initial request is sufficient for granting the requested access, the AS MAY return the positive results immediately in its Section 3 with access tokens and subject information.

If the AS determines that additional runtime authorization is required, the AS can either deny the request outright or use a number of means at its disposal to gather that authorization from the appropriate ROs, including for example:

* starting interaction with the end user facilitated by the client software, such as a redirection or user code

* challenging the client instance through a challenge-response mechanism

* requesting that the client instance present specific additional information, such as a user's credential or an assertion

* contacting a RO through an out-of-band mechanism, such as a push notification

* contacting an auxiliary software process through an out-of-band mechanism, such as querying a digital wallet

The authorization and consent gathering process in GNAP is left deliberately flexible to allow for a wide variety of different deployments, interactions, and methodologies. In this process, the AS can gather consent from the RO as necessitated by the access that has been requested. The AS can sometimes determine which RO needs to consent based on what has been requested by the client instance, such as a specific RS record, an identified user, or a request requiring specific access such as approval by an administrator. If the AS has a means of contacting the RO directly, it could do so without involving the client instance in its consent gathering process. For example, the AS could push a notification to a known RO and have the RO approve the pending request asynchronously. These interactions can be through an interface of the AS itself (such as a hosted web page), through another application (such as something installed on the RO's device), through a messaging fabric, or any other means. When interacting with an RO, the AS can do anything it needs to determine the authorization of the requested grant, including:

* authenticate the RO, through a local account or some other means such as federated login

* validate the RO through presentation of claims, attributes, or other information

* prompt the RO for consent for the requested delegation

* describe to the RO what information is being released, to whom, and for what purpose

* provide warnings to the RO about potential attacks or negative effects of allowing the information

* allow the RO to modify the client instance's requested access, including limiting or expanding that access

* provide the RO with artifacts such as receipts to facilitate an audit trail of authorizations

* allow the RO to deny the requested delegation

The AS is also allowed to request authorization from more than one RO, if the AS deems fit. For example, a medical record might need to be released by both an attending nurse and a physician, or both owners of a bank account need to sign off on a transfer request. Alternatively, the AS could require N of M possible RO's to approve a

given request in order. The AS could also determine that the end user is not the appropriate RO for a given request and reach out to the appropriate RO asynchronously. The details of determining which RO's are required for a given request are out of scope for this specification.

The client instance can also indicate that it is capable of facilitating interaction with the end user, another party, or another piece of software through its interaction start (Section 2.5.1) request. In many cases, the end user is delegating their own access as RO to the client instance. Here, the AS needs to determine the identity of the end user and will often need to interact directly with the end user to determine their status as an RO and collect their consent. If the AS has determined that authorization is required and the AS can support one or more of the requested interaction start methods, the AS returns the associated interaction start responses (Section 3.3). The client instance SHOULD initiate one or more of these interaction methods (Section 4.1) in order to facilitate the granting of the request. If more than one interaction start method is available, the means by which the client chooses which methods to follow is out of scope of this specification. The client instance MUST use each interaction method once at most.

After starting interaction, the client instance can then make a continuation request (Section 5) either in response to a signal indicating the finish of the interaction (Section 4.2), through polling, or through some other method defined by an extension of this specification.

If the AS and client instance have not reached a state where the delegation can be granted, the AS and client instance can repeat the interaction process as long as the AS supplies the client instance with continuation information (Section 3.1) to facilitate the ongoing requests.

4.1. Interaction Start Methods

To initiate an interaction start method indicated by the interaction start responses (Section 3.3) from the AS, the client instance follows the steps defined by that interaction method. The actions of the client instance required for the interaction start modes defined in this specification are described in the following sections.

4.1.1. Interaction at a Redirected URI

When the end user is directed to an arbitrary URI through the "redirect" (Section 3.3.1) mode, the client instance facilitates opening the URI through the end user's web browser. The client instance could launch the URI through the system browser, provide a clickable link, redirect the user through HTTP response codes, or display the URI in a form the end user can use to launch such as a multidimensional barcode. With this method, it is common (though not required) for the RO to be the same party as the end-user, since the client instance has to communicate the redirection URI to the end-user.

In many cases, the URI indicates a web page hosted at the AS, allowing the AS to authenticate the end user as the RO and interactively provide consent. If the URI is hosted by the AS, the AS MUST determine the grant request being referenced from the URL value itself. If the URL cannot be associated with a currently active request, the AS MUST display an error to the RO and MUST NOT attempt to redirect the RO back to any client instance even if a redirect finish method is supplied (Section 2.5.2.1). If the URI is not hosted by the AS directly, the means of communication between the AS and this URI are out of scope for this specification.

The client instance MUST NOT modify the URI when launching it, in particular the client instance MUST NOT add any parameters to the URI. The URI MUST be reachable from the end user's browser, though the URI MAY be opened on a separate device from the client instance itself. The URI MUST be accessible from an HTTP GET request and MUST be protected by HTTPS or equivalent means.

4.1.2. Interaction at the User Code URI

When the end user is directed to enter a short code through the "user_code" (Section 3.3.3) mode, the client instance communicates the user code to the end-user and directs the end user to enter that code at an associated URI. This mode is used when the client instance is not able to facilitate launching an arbitrary URI. The associated URI could be statically configured with the client instance or communicated in the response from the AS, but the client instance communicates that URL to the end user. As a consequence, these URIs SHOULD be short.

In many cases, the URI indicates a web page hosted at the AS, allowing the AS to authenticate the end user as the RO and interactively provide consent. If the URI is hosted by the AS, the AS MUST determine the grant request being referenced from the user code. If the user code cannot be associated with a currently active

request, the AS MUST display an error to the RO and MUST NOT attempt to redirect the RO back to any client instance even if a redirect finish method is supplied (Section 2.5.2.1). If the interaction component at the user code URI is not hosted by the AS directly, the means of communication between the AS and this URI, including communication of the user code itself, are out of scope for this specification.

When the RO enters this code at the user code URI, the AS MUST uniquely identify the pending request that the code was associated with. If the AS does not recognize the entered code, the interaction component MUST display an error to the user. If the AS detects too many unrecognized code enter attempts, the interaction component SHOULD display an error to the user and MAY take additional actions such as slowing down the input interactions. The user should be warned as such an error state is approached, if possible.

The client instance MUST NOT modify the URI when launching it, in particular the client instance MUST NOT add any parameters to the URI. The user code URI MUST be reachable from the end user's browser, though the URI is usually be opened on a separate device from the client instance itself. The URI MUST be accessible from an HTTP GET request and MUST be protected by HTTPS or equivalent means.

4.1.3. Interaction through an Application URI

When the client instance is directed to launch an application through the "app" (Section 3.3.2) mode, the client launches the URL as appropriate to the system, such as through a deep link or custom URI scheme registered to a mobile application. The means by which the AS and the launched application communicate with each other and perform any of the required actions are out of scope for this specification.

4.2. Post-Interaction Completion

If an interaction "finish" (Section 3.3.4) method is associated with the current request, the AS MUST follow the appropriate method at upon completion of interaction in order to signal the client instance to continue, except for some limited error cases discussed below. If a finish method is not available, the AS SHOULD instruct the RO to return to the client instance upon completion.

The AS MUST create an interaction reference and associate that reference with the current interaction and the underlying pending request. This interaction reference value MUST be sufficiently random so as not to be guessable by an attacker. The interaction reference MUST be one-time-use to prevent interception and replay attacks.

The AS MUST calculate a hash value based on the client instance and AS nonces and the interaction reference, as described in Section 4.2.3. The client instance will use this value to validate the "finish" call.

The AS MUST send the hash and interaction reference based on the interaction finish mode as described in the following sections.

Note that the "finish" method still occurs in many error cases, such as when the RO has denied access. This pattern allows the client instance to potentially recover from the error state by modifying its request or providing additional information directly to the AS in a continuation request. The AS MUST NOT follow the "finish" method in the following circumstances:

* The AS has determined that any URIs involved with the finish method are dangerous or blocked.

* The AS cannot determine which ongoing grant request is being referenced.

* The ongoing grant request has been cancelled or otherwise blocked.

4.2.1. Completing Interaction with a Browser Redirect to the Callback URI

When using the "redirect" interaction finish method (Section 3.3.4), the AS signals to the client instance that interaction is complete and the request can be continued by directing the RO (in their browser) back to the client instance's redirect URL sent in the callback request (Section 2.5.2.1).

The AS secures this redirect by adding the hash and interaction reference as query parameters to the client instance's redirect URL.

hash REQUIRED. The interaction hash value as described in Section 4.2.3.

interact_ref REQUIRED. The interaction reference generated for this interaction.

The means of directing the RO to this URL are outside the scope of this specification, but common options include redirecting the RO from a web page and launching the system browser with the target URL.

```
https://client.example.net/return/123455\
  ?hash=p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jZQ1R2\
    HZT8BOWYHcLmObM7XHPAdJzTZMtKBsaraJ64A\
  &interact_ref=4IFWWIKYBC2PQ6U56NL1
```

When receiving the request, the client instance MUST parse the query parameters to calculate and validate the hash value as described in Section 4.2.3. If the hash validates, the client instance sends a continuation request to the AS as described in Section 5.1 using the interaction reference value received here.

4.2.2. Completing Interaction with a Direct HTTP Request Callback

When using the "callback" interaction mode (Section 3.3.4) with the "push" method, the AS signals to the client instance that interaction is complete and the request can be continued by sending an HTTP POST request to the client instance's callback URL sent in the callback request (Section 2.5.2.2).

The entity message body is a JSON object consisting of the following two fields:

hash (string) REQUIRED. The interaction hash value as described in Section 4.2.3.

interact_ref (string) REQUIRED. The interaction reference generated for this interaction.

```
POST /push/554321 HTTP/1.1
Host: client.example.net
Content-Type: application/json

{
  "hash": "p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jZQ1R\
    2HZT8BOWYHcLmObM7XHPAdJzTZMtKBsaraJ64A",
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

When receiving the request, the client instance MUST parse the JSON object and validate the hash value as described in Section 4.2.3. If the hash validates, the client instance sends a continuation request to the AS as described in Section 5.1 using the interaction reference value received here.

4.2.3. Calculating the interaction hash

The "hash" parameter in the request to the client instance's callback URL ties the front channel response to an ongoing request by using values known only to the parties involved. This security mechanism allows the client instance to protect itself against several kinds of session fixation and injection attacks. The AS MUST always provide this hash, and the client instance MUST validate the hash when received.

To calculate the "hash" value, the party doing the calculation creates a hash string by concatenating the following values in the following order using a single newline ("\\n") character to separate them:

* the "nonce" value sent by the client instance in the interaction "finish" section of the initial request (Section 2.5.2)

* the AS's nonce value from the interaction finish response (Section 3.3.4)

* the "interact_ref" returned from the AS as part of the interaction finish method (Section 4.2)

* the grant endpoint URL the client instance used to make its initial request (Section 2)

There is no padding or whitespace before or after any of the lines, and no trailing newline character.

```
VJLO6A4CAYLBXHTR0KRO
MBDOFXG4Y5CVJCX821LH
4IFWWIKYBC2PQ6U56NL1
https://server.example.com/tx
```

The party then hashes this string with the appropriate algorithm based on the "hash_method" parameter of the "callback". If the "hash_method" value is not present in the client instance's request, the algorithm defaults to "sha3".

[[See issue #56 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/56)]]

4.2.3.1. SHA3-512

The "sha3" hash method consists of hashing the input string with the 512-bit SHA3 algorithm. The byte array is then encoded using URL Safe Base64 with no padding. The resulting string is the hash value.

```
p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jZQ1R2HZT8BOWYHcLmObM\
  7XHPAdJzTZMtKBsaraJ64A
```

4.2.3.2. SHA2-512

The "sha2" hash method consists of hashing the input string with the 512-bit SHA2 algorithm. The byte array is then encoded using URL Safe Base64 with no padding. The resulting string is the hash value.

```
62SbcD3Xs7L40rjgALA-ymQujoh2LB2hPJyX9vlcr1H6ecChZ8BNKkG_HrOKP_Bp\
  j84rh4mC9aE9x7HPBFcIHw
```

5. Continuing a Grant Request

While it is possible for the AS to return a Section 3 with all the client instance's requested information (including access tokens (Section 3.2) and direct user information (Section 3.4)), it's more common that the AS and the client instance will need to communicate several times over the lifetime of an access grant. This is often part of facilitating interaction (Section 4), but it could also be used to allow the AS and client instance to continue negotiating the parameters of the original grant request (Section 2).

To enable this ongoing negotiation, the AS provides a continuation API to the client software. The AS returns a "continue" field in the response (Section 3.1) that contains information the client instance needs to access this API, including a URI to access as well as an access token to use during the continued requests.

The access token is initially bound to the same key and method the client instance used to make the initial request. As a consequence, when the client instance makes any calls to the continuation URL, the client instance MUST present the access token as described in Section 7.2 and present proof of the client instance's key (or its most recent rotation) by signing the request as described in Section 7.3. The AS MUST validate all keys presented by the client instance or referenced in an ongoing request for each call within that request.

[[See issue #85 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/85)]]

For example, here the client instance makes a POST request to a unique URI and signs the request with HTTP Message Signatures:

```
POST /continue/KSKUOMUKM HTTP/1.1
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Host: server.example.com
Signature-Input: sig1=...
Signature: sig1=...
```

The AS MUST be able to tell from the client instance's request which specific ongoing request is being accessed, using a combination of the continuation URL, the provided access token, and the client instance identified by the key signature. If the AS cannot determine a single active grant request to map the continuation request to, the AS MUST return an error.

The ability to continue an already-started request allows the client instance to perform several important functions, including presenting additional information from interaction, modifying the initial request, and getting the current state of the request.

All requests to the continuation API are protected by this bound access token. For example, here the client instance makes a POST request to a stable continuation endpoint URL with the interaction reference (Section 5.1), includes the access token, and signs with HTTP Message Signatures:

```
POST /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

If a "wait" parameter was included in the continuation response (Section 3.1), the client instance MUST NOT call the continuation URI prior to waiting the number of seconds indicated. If no "wait" period is indicated, the client instance SHOULD wait at least 5 seconds. If the client instance does not respect the given wait period, the AS MUST return an error. [[See issue #86 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/86)]]

The response from the AS is a JSON object and MAY contain any of the fields described in Section 3, as described in more detail in the sections below.

If the AS determines that the client instance can make a further continuation request, the AS MUST include a new "continue" response (Section 3.1). The new "continue" response MUST include a bound access token as well, and this token SHOULD be a new access token, invalidating the previous access token. If the AS does not return a new "continue" response, the client instance MUST NOT make an additional continuation request. If a client instance does so, the AS MUST return an error. [[See issue #87 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/87)]]

For continuation functions that require the client instance to send a message body, the body MUST be a JSON object.

5.1. Continuing After a Completed Interaction

When the AS responds to the client instance's "finish" method as in Section 4.2.1, this response includes an interaction reference. The client instance MUST include that value as the field "interact_ref" in a POST request to the continuation URI.

```
POST /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
  "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

Since the interaction reference is a one-time-use value as described in Section 4.2.1, if the client instance needs to make additional continuation calls after this request, the client instance MUST NOT include the interaction reference. If the AS detects a client instance submitting the same interaction reference multiple times, the AS MUST return an error and SHOULD invalidate the ongoing request.

The Section 3 MAY contain any newly-created access tokens (Section 3.2) or newly-released subject claims (Section 3.4). The response MAY contain a new "continue" response (Section 3.1) as described above. The response SHOULD NOT contain any interaction responses (Section 3.3). [[See issue #89 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/89)]]

For example, if the request is successful in causing the AS to issue access tokens and release opaque subject claims, the response could look like this:

```
{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
    },
    "subject": {
        "sub_ids": [ {
           "format": "opaque",
           "id": "J2G8G8O4AZ"
        } ]
    }
}
```

With this example, the client instance can not make an additional continuation request because a "continue" field is not included.

[[See issue #88 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/88)]]

5.2. Continuing During Pending Interaction

When the client instance does not include a "finish" parameter, the client instance will often need to poll the AS until the RO has authorized the request. To do so, the client instance makes a POST request to the continuation URI as in Section 5.1, but does not include a message body.

```
POST /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
```

The Section 3 MAY contain any newly-created access tokens (Section 3.2) or newly-released subject claims (Section 3.4). The response MAY contain a new "continue" response (Section 3.1) as described above. If a "continue" field is included, it SHOULD include a "wait" field to facilitate a reasonable polling rate by the client instance. The response SHOULD NOT contain interaction responses (Section 3.3).

For example, if the request has not yet been authorized by the RO, the AS could respond by telling the client instance to make another continuation request in the future. In this example, a new, unique access token has been issued for the call, which the client instance will use in its next continuation request.

```
{
    "continue": {
        "access_token": {
            "value": "33OMUKMKSKU80UPRY5NM"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    }
}
```

[[See issue #90 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/90)]]

[[See issue #91 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/91)]]

If the request is successful in causing the AS to issue access tokens and release subject claims, the response could look like this example:

```
{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
    },
    "subject": {
        "sub_ids": [ {
           "format": "opaque",
           "id": "J2G8G8O4AZ"
        } ]
    }
}
```

5.3. Modifying an Existing Request

The client instance might need to modify an ongoing request, whether or not tokens have already been issued or claims have already been released. In such cases, the client instance makes an HTTP PATCH request to the continuation URI and includes any fields it needs to modify. Fields that aren't included in the request are considered unchanged from the original request.

The client instance MAY include the "access_token" and "subject" fields as described in Section 2.1 and Section 2.2. Inclusion of these fields override any values in the initial request, which MAY trigger additional requirements and policies by the AS. For example, if the client instance is asking for more access, the AS could require additional interaction with the RO to gather additional consent. If the client instance is asking for more limited access, the AS could determine that sufficient authorization has been granted to the client instance and return the more limited access rights immediately. [[See issue #92 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/92)]]

The client instance MAY include the "interact" field as described in Section 2.5. Inclusion of this field indicates that the client instance is capable of driving interaction with the RO, and this field replaces any values from a previous request. The AS MAY respond to any of the interaction responses as described in Section 3.3, just like it would to a new request.

The client instance MAY include the "user" field as described in Section 2.4 to present new assertions or information about the end-user. [[See issue #93 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/93)]]

The client instance MUST NOT include the "client" section of the request. [[See issue #94 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/94)]]

The client instance MAY include post-interaction responses such as described in Section 5.1. [[See issue #95 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/95)]]

Modification requests MUST NOT alter previously-issued access tokens. Instead, any access tokens issued from a continuation are considered new, separate access tokens. The AS MAY revoke existing access tokens after a modification has occurred. [[See issue #96 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/96)]]

If the modified request can be granted immediately by the AS, the Section 3 MAY contain any newly-created access tokens (Section 3.2) or newly-released subject claims (Section 3.4). The response MAY contain a new "continue" response (Section 3.1) as described above. If interaction can occur, the response SHOULD contain interaction responses (Section 3.3) as well.

For example, a client instance initially requests a set of resources using references:

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
    "access_token": {
        "access": [
            "read", "write"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    },
    "client": "987YHGRT56789IOLK"
}
```

Access is granted by the RO, and a token is issued by the AS. In its final response, the AS includes a "continue" field, which includes a separate access token for accessing the continuation API:

```
{
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    },
    "access_token": {
        "value": "RP1LT0-OS9M2P_R64TB",
        "access": [
            "read", "write"
        ]
    }
}
```

This "continue" field allows the client instance to make an eventual continuation call. In the future, the client instance realizes that it no longer needs "write" access and therefore modifies its ongoing request, here asking for just "read" access instead of both "read" and "write" as before.

```
PATCH /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
    "access_token": {
        "access": [
            "read"
        ]
    }
    ...
}
```

The AS replaces the previous "access" from the first request, allowing the AS to determine if any previously-granted consent already applies. In this case, the AS would likely determine that reducing the breadth of the requested access means that new access tokens can be issued to the client instance. The AS would likely revoke previously-issued access tokens that had the greater access rights associated with them, unless they had been issued with the "durable" flag.

```
{
    "continue": {
        "access_token": {
            "value": "M33OMUK80UPRY5NMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    },
    "access_token": {
        "value": "0EVKC7-2ZKwZM_6N760",
        "access": [
            "read"
        ]
    }
}
```

For another example, the client instance initially requests read-only access but later needs to step up its access. The initial request could look like this example.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
    "access_token": {
        "access": [
            "read"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    },
    "client": "987YHGRT56789IOLK"
}
```

Access is granted by the RO, and a token is issued by the AS. In its final response, the AS includes a "continue" field:

```
{
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 30
    },
    "access_token": {
        "value": "RP1LT0-OS9M2P_R64TB",
        "access": [
            "read"
        ]
    }
}
```

This allows the client instance to make an eventual continuation call. The client instance later realizes that it now needs "write" access in addition to the "read" access. Since this is an expansion of what it asked for previously, the client instance also includes a new interaction section in case the AS needs to interact with the RO again to gather additional authorization. Note that the client instance's nonce and callback are different from the initial request. Since the original callback was already used in the initial exchange, and the callback is intended for one-time-use, a new one needs to be included in order to use the callback again.

[[See issue #97 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/97)]]

```
PATCH /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
    "access_token": {
        "access": [
            "read", "write"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/654321",
            "nonce": "K82FX4T4LKLTI25DQFZC"
        }
    }
}
```

From here, the AS can determine that the client instance is asking for more than it was previously granted, but since the client instance has also provided a mechanism to interact with the RO, the AS can use that to gather the additional consent. The protocol continues as it would with a new request. Since the old access tokens are good for a subset of the rights requested here, the AS might decide to not revoke them. However, any access tokens granted after this update process are new access tokens and do not modify the rights of existing access tokens.

5.4. Canceling a Grant Request

If the client instance wishes to cancel an ongoing grant request, it makes an HTTP DELETE request to the continuation URI.

```
DELETE /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
```

If the request is successfully cancelled, the AS responds with an HTTP 202. The AS SHOULD revoke all associated access tokens.

6. Token Management

If an access token response includes the "manage" parameter as described in Section 3.2.1, the client instance MAY call this URL to manage the access token with any of the actions defined in the following sections. Other actions are undefined by this specification.

The access token being managed acts as the access element for its own management API. The client instance MUST present proof of an appropriate key along with the access token.

If the token is sender-constrained (i.e., not a bearer token), it MUST be sent with the appropriate binding for the access token (Section 7.2).

If the token is a bearer token, the client instance MUST present proof of the same key identified in the initial request (Section 2.3) as described in Section 7.3.

The AS MUST validate the proof and assure that it is associated with either the token itself or the client instance the token was issued to, as appropriate for the token’s presentation type.

6.1. Rotating the Access Token

The client instance makes an HTTP POST to the token management URI, sending the access token in the appropriate header and signing the request with the appropriate key.

```
POST /token/PRY5NM33OM4TB8N6BW7OZB8CDFONP219RP1L HTTP/1.1
Host: server.example.com
Authorization: GNAP OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...
```

The AS validates that the token presented is associated with the management URL, that the AS issued the token to the given client instance, and that the presented key is appropriate to the token.

If the access token has expired, the AS SHOULD honor the rotation request to the token management URL since it is likely that the client instance is attempting to refresh the expired token. To support this, the AS MAY apply different lifetimes for the use of the token in management vs. its use at an RS. An AS MUST NOT honor a rotation request for an access token that has been revoked, either by the AS or by the client instance through the token management URI (Section 6.2).

If the token is validated and the key is appropriate for the request, the AS MUST invalidate the current access token associated with this URL, if possible, and return a new access token response as described in Section 3.2.1, unless the "multi_token" flag is specified in the request. The value of the access token MUST NOT be the same as the current value of the access token used to access the management API. The response MAY include an updated access token management URL as well, and if so, the client instance MUST use this new URL to manage the new access token. [[See issue #101 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/101)]]

[[See issue #102 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/102)]]

```
{
    "access_token": {
        "value": "FP6A8H6HY37MH13CK76LBZ6Y1UADG6VEUPEER5H2",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [
            {
                "type": "photo-api",
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            },
            "read", "dolphin-metadata"
        ]
    }
}
```

[[See issue #103 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/103)]]

6.2. Revoking the Access Token

If the client instance wishes to revoke the access token proactively, such as when a user indicates to the client instance that they no longer wish for it to have access or the client instance application detects that it is being uninstalled, the client instance can use the token management URI to indicate to the AS that the AS should invalidate the access token for all purposes.

The client instance makes an HTTP DELETE request to the token management URI, presenting the access token and signing the request with the appropriate key.

```
DELETE /token/PRY5NM33OM4TB8N6BW7OZB8CDFONP219RP1L HTTP/1.1
Host: server.example.com
Authorization: GNAP OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0
Signature-Input: sig1=...
Signature: sig1=...
```

If the key presented is associated with the token (or the client instance, in the case of a bearer token), the AS MUST invalidate the access token, if possible, and return an HTTP 204 response code.

204 No Content

Though the AS MAY revoke an access token at any time for any reason, the token management function is specifically for the client instance's use. If the access token has already expired or has been revoked through other means, the AS SHOULD honor the revocation request to the token management URL as valid, since the end result is still the token not being usable.

7. Securing Requests from the Client Instance

In GNAP, the client instance secures its requests to the AS and RS by presenting an access token, presenting proof of a key that it possesses, or both an access token and key proof together.

* When an access token is used with a key proof, this is a bound token request. This type of request is used for calls to the RS as well as the AS during negotiation.

* When a key proof is used with no access token, this is a non-authorized signed request. This type of request is used for calls to the AS to initiate a negotiation.

* When an access token is used with no key proof, this is a bearer token request. This type of request is used only for calls to the RS, and only with access tokens that are not bound to any key as described in Section 3.2.1.

* When neither an access token nor key proof are used, this is an unsecured request. This type of request is used optionally for calls to the RS as part of an RS-first discovery process as described in Section 9.1.

7.1. Key Formats

Several different places in GNAP require the presentation of key material by value. Proof of this key material MUST be bound to a request, the nature of which varies with the location in the protocol the key is used. For a key used as part of a client instance's initial request in Section 2.3, the key value is the client instance's public key, and proof of that key MUST be presented in that request. For a key used as part of an access token response in Section 3.2.1, the proof of that key MUST be used when presenting the access token.

A key presented by value MUST be a public key in at least one supported format. If a key is sent in multiple formats, all the key format values MUST be equivalent. Note that while most formats present the full value of the public key, some formats present a value cryptographically derived from the public key.

proof (string) The form of proof that the client instance will use when presenting the key. The valid values of this field and the processing requirements for each are detailed in Section 7.3. The "proof" field is REQUIRED.

jwk (object) The public key and its properties represented as a JSON Web Key [RFC7517]. A JWK MUST contain the "alg" (Algorithm) and "kid" (Key ID) parameters. The "alg" parameter MUST NOT be "none". The "x5c" (X.509 Certificate Chain) parameter MAY be used to provide the X.509 representation of the provided public key.

cert (string) PEM serialized value of the certificate used to sign the request, with optional internal whitespace per [RFC7468]. The PEM header and footer are optionally removed.

cert#S256 (string) The certificate thumbprint calculated as per OAuth-MTLS [RFC8705] in base64 URL encoding. Note that this format does not include the full public key.

Additional key formats are defined in a registry TBD (Section 11).

This non-normative example shows a single key presented in multiple formats. This example key is intended to be used with the HTTP Message Signatures ({{httpsig-binding}}) proofing mechanism, as indicated by the "httpsig" value of the "proof" field.

```
"key": {
    "proof": "httpsig",
    "jwk": {
                "kty": "RSA",
                "e": "AQAB",
                "kid": "xyz-1",
                "alg": "RS256",
                "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8xY..."
    },
    "cert": "MIIEHDCCAwSgAwIBAgIBATANBgkqhkiG9w0BAQsFA..."
}
```

7.1.1. Key References

Keys in GNAP can also be passed by reference such that the party receiving the reference will be able to determine the appropriate keying material for use in that part of the protocol.

```
"key": "S-P4XJQ_RYJCRTSU1.63N3E"
```

Keys referenced in this manner MAY be shared symmetric keys. The key reference MUST NOT contain any unencrypted private or shared symmetric key information.

Keys referenced in this manner MUST be bound to a single proofing mechanism.

The means of dereferencing this value are out of scope for this specification.

7.2. Presenting Access Tokens

The method the client instance uses to send an access token depends on whether the token is bound to a key, and if so which proofing method is associated with the key. This information is conveyed in the "bound" and "key" parameters in the single (Section 3.2.1) and multiple access tokens (Section 3.2.2) responses.

If the "flags" field does not contain the "bearer" flag and the "key" is absent, the access token MUST be sent using the same key and proofing mechanism that the client instance used in its initial request (or its most recent rotation).

If the "flags" field does not contain the "bearer" flag and the "key" value is an object as described in Section 7.1, the access token MUST be sent using the key and proofing mechanism defined by the value of the "proof" field within the key object.

The access token MUST be sent using the HTTP "Authorization" request header field and the "GNAP" authorization scheme along with a key proof as described in Section 7.3 for the key bound to the access token. For example, an "httpsig"-bound access token is sent as follows:

```
Authorization: GNAP OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0
Signature-Input: sig1=(authorization);...
Signature: sig1=...
```

If the "flags" field contains the "bearer" flag, the access token is a bearer token that MUST be sent using the "Authorization Request Header Field" method defined in [RFC6750].

Authorization: Bearer OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0

The "Form-Encoded Body Parameter" and "URI Query Parameter" methods of [RFC6750] MUST NOT be used.

[[See issue #104 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/104)]]

The client software MUST reject as an error a situation where the "flags" field contains the "bearer" flag and the "key" field is present with any value.

7.3. Proving Possession of a Key with a Request

Any keys presented by the client instance to the AS or RS MUST be validated as part of the request in which they are presented. The type of binding used is indicated by the proof parameter of the key object in Section 7.1. Values defined by this specification are as follows:

httpsig HTTP Signing signature header

mtls Mutual TLS certificate verification

jwsd A detached JWS signature header

jws Attached JWS payload

Additional proofing methods are defined by a registry TBD (Section 11).

All key binding methods used by this specification MUST cover all relevant portions of the request, including anything that would change the nature of the request, to allow for secure validation of the request. Relevant aspects include the URI being called, the HTTP method being used, any relevant HTTP headers and values, and the HTTP message body itself. The verifier of the signed message MUST validate all components of the signed message to ensure that nothing has been tampered with or substituted in a way that would change the nature of the request. Key binding method definitions SHOULD enumerate how these requirements are fulfilled.

When a key proofing mechanism is bound to an access token, the key being presented MUST be the key associated with the access token and the access token MUST be covered by the signature method of the proofing mechanism.

The key binding methods in this section MAY be used by other components making calls as part of GNAP, such as the extensions allowing the RS to make calls to the AS defined in {{I-D.ietf-gnap-resource-servers}}. To facilitate this extended use, the sections below are defined in generic terms of the "sender" and "verifier" of the HTTP message. In the core functions of GNAP, the "sender" is the client instance and the "verifier" is the AS or RS, as appropriate.

When used for delegation in GNAP, these key binding mechanisms allow the AS to ensure that the keys presented by the client instance in the initial request are in control of the party calling any follow-up or continuation requests. To facilitate this requirement, the continuation response (Section 3.1) includes an access token bound to the client instance's key (Section 2.3), and that key (or its most recent rotation) MUST be proved in all continuation requests Section 5. Token management requests Section 6 are similarly bound to either the access token's own key or, in the case of bearer tokens, the client instance's key.

[[See issue #105 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/105)]]

In the following sections, unless otherwise noted, the "RS256" JOSE Signature Algorithm is applied using the following RSA key (presented here in JWK format):

```
{
    "kid": "gnap-rsa",
    "p": "xS4-YbQ0SgrsmcA7xDzZKuVNxJe3pCYwdAe6efSy4hdDgF9-vhC5gjaRk\
        i1wWuERSMW4Tv44l5HNrL-Bbj_nCJxr_HAOaesDiPn2PnywwEfg3Nv95Nn-\
        eilhqXRaW-tJKEMjDHu_fmJBeemHNZI412gBnXdGzDVo22dvYoxd6GM",
    "kty": "RSA",
    "q": "rVdcT_uy-CD0GKVLGpEGRR7k4JO6Tktc8MEHkC6NIFXihk_6vAIOCzCD6\
        LMovMinOYttpRndKoGTNdJfWlDFDScAs8C5n2y1STCQPRximBY-bw39-aZq\
        JXMxOLyPjzuVgiTOCBIvLD6-8-mvFjXZk_eefD0at6mQ5qV3U1jZt88",
    "d": "FHlhdTF0ozTliDxMBffT6aJVKZKmbbFJOVNten9c3lXKB3ux3NAb_D2dB\
        7inp9EV23oWrDspFtvCvD9dZrXgRKMHofkEpo_SSvBZfgtH-OTkbY_TqtPF\
        FLPKAw0JX5cFPnn4Q2xE4n-dQ7tpRCKl59vZLHBrHShr90zqzFp0AKXU5fj\
        b1gC9LPwsFA2Fd7KXmI1drQQEVq9R-o18Pnn4BGQNQNjO_VkcJTiBmEIVT_\
        KJRPdpVJAmbgnYWafL_hAfeb_dK8p85yurEVF8nCK5oO3EPrqB7IL4UqaEn\
        5Sl3u0j8x5or-xrrAoNz-gdOv7ONfZY6NFoa-3f8q9wBAHUuQ",
    "e": "AQAB",
    "qi": "ogpNEkDKg22Rj9cDV_-PJBZaXMk66Fp557RT1tafIuqJRHEufSOYnsto\
        bWPJ0gHxv1gVJw3gm-zYvV-wTMNgr2wVsBSezSJjPSjxWZtmT2z68W1DuvK\
        kZy15vz7Jd85hmDlriGcXNCoFEUsGLWkpHH9RwPIzguUHWmTt8y0oXyI",
    "dp": "dvCKGI2G7RLh3WyjoJ_Dr6hZ3LhXweB3YcY3qdD9BnxZ71mrLiMQg4c_\
        EBnwqCETN_5sStn2cRc2JXnvLP3G8t7IFKHTT_i_TSTacJ7uT04MSa053Y3\
        RfwbvLjRNPR0UKAE3ZxROUoIaVNuU_6-QMf8-2ilUv2GIOrCN87gP_Vk",
    "alg": "RS256",
    "dq": "iMZmELaKgT9_W_MRT-UfDWtTLeFjIGRW8aFeVmZk9R7Pnyt8rNzyN-IQ\
        M40ql8u8J6vc2GmQGfokLlPQ6XLSCY68_xkTXrhoU1f-eDntkhP7L6XawSK\
        Onv5F2H7wyBQ75HUmHTg8AK2B_vRlMyFKjXbVlzKf4kvqChSGEz4IjQ",
    "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8BfYdHsFzAt\
        YKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZGYX\
        jHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZx\
        e0jRETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0\
        bunS0K3bA_3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kO\
        zywzwPTuq-cVQDyEN7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
}
```


7.3.1. HTTP Message Signing

This method is indicated by "httpsig" in the "proof" field. The sender creates an HTTP Message Signature as described in [I-D.ietf-httpbis-message-signatures].

The covered content of the signature MUST include the following:

@request-target: the target of the HTTP request

digest: The Digest header as defined in [RFC3230]. When the request message has a body, the signer MUST calculate this header value and the verifier MUST validate this header.

When the request is bound to an access token, the covered content MUST also include:

authorization: The Authorization header used to present the access token as discussed in Section 7.2.

Other covered content MAY also be included.

If the signer’s key presented is a JWK, the "keyid" parameter of the signature MUST be set to the "kid" value of the JWK, the signing algorithm used MUST be the JWS algorithm denoted by the key’s "alg" field, and the explicit "alg" signature parameter MUST NOT be included.

In this example, the message body is the following JSON object:

```
{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "proof": "httpsig",
      "key": {
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "RS256",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    }
}
```

This body is hashed for the Digest header using SHA-256 into the following encoded value:

```
SHA-256=98QzyNVYpdgTrWBKpC4qFSCmmR+CrwwvUoiaDCSjKxw=
```

The HTTP message signature input string is calculated to be the following:

```
"@request-target": post /gnap
"host": server.example.com
"content-type": application/json
"digest": SHA-256=98QzyNVYpdgTrWBKpC4qFSCmmR+CrwwvUoiaDCSjKxw=
"content-length": 986
"@signature-params": ("@request-target" "host" "content-type" \
  "digest" "content-length");created=1618884475;keyid="gnap-rsa"
```

This leads to the following full HTTP message request:

```
POST /gnap HTTP/1.1
Host: server.example.com
Content-Type: application/json
Content-Length: 986
Digest: SHA-256=98QzyNVYpdgTrWBKpC4qFSCmmR+CrwwvUoiaDCSjKxw=
Signature-Input: sig1=("@request-target" "host" "content-type" \
  "digest" "content-length");created=1618884475;keyid="gnap-rsa"
Signature: \
  sig1=:axj8FLOvEWBcwh+Xk6VTTKXxqo4XNygleTDJ8h3ZJfi1sSmWrRtyo9RG/dc\
  miZmdszRjWbg+/ixVZpA4BL3AOwEOxxtmHAXNB8uJ0I3tfbs6Suyk4sEo8zPr+MJq\
  MjxdJEUgAQAy2AH+wg5a7CKq4IdLTulFK9njUIeG7MygHumeiumM3DbDQAHgF46dV\
  q5UC6KJnqhGM1rFC128jd2D0sgWKCUgKGCHtfR159zfKWcEO9krsLoOnCdTzm1UyD\
  DMjkIjqeN/1j8PdMJaRAwV4On079O0DVu6bl1jVtkzo/e/ZmwPr/X436V4xiw/hZt\
  w4sfNsSbmsT0+UAQ20X/xaw==:
```

```
{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "proof": "httpsig",
      "key": {
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "RS256",
```

```
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
    YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
    YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
    ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
    3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
    N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
          }
        }
        "display": {
          "name": "My Client Display Name",
          "uri": "https://client.foo/"
        },
      }
  }
```

If the HTTP Message includes a message body, the verifier MUST calculate and verify the value of the "Digest" header. The verifier MUST ensure that the signature includes all required covered content. The verifier MUST validate the signature against the expected key of the signer.

7.3.2. Mutual TLS

This method is indicated by "mtls" in the "proof" field. The signer presents its TLS client certificate during TLS negotiation with the verifier.

In this example, the certificate is communicated to the application through the "Client-Cert" header from a TLS reverse proxy, leading to the following full HTTP request message:

```
POST /gnap HTTP/1.1
Host: server.example.com
Content-Type: application/jose
Content-Length: 1567
Client-Cert: \
  MIIC6jCCAdKgAwIBAgIGAXjw74xPMA0GCSqGSIb3DQEBCwUAMDYxNDAyBgNVBAMM \
  K05JWU15QmpzRGp5QkM5UDUzN0Q2SVR6a3BEOE50UmppOXlhcEV6QzY2bVEwHhcN \
  MjEwNDIwMjAxODU0WhcNMjIwMjE0MjAxODU0WjA2MTQwMgYDVQQDDCtOSVlNeUJq \
  c0RqeUJDOVA1MzdENklUemtwRDhOdFJqaTl5YXBFekM2Nm1RMIIBIjANBgkqhkiG \
  9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhYOJ+XOKISdMMShn/G4W9m20mT0VWtQBsmBB \
  kI2cmRt4Ai8BfYdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8I \
  kZ8NMwSrcUIBZGYXjHpwjzvfGvXH/5KJlnR3/uRUp4Z4Ujk2bCaKegDn11V2vxE4 \
  1hqaPUnhRZxe0jRETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo+ \
  uv4BC0bunS0K3bA/3UgVp7zBlQFoFnLTO2uWp/muLEWGl67gBq9MO3brKXfGhi3k \
  OzywzwPTuq+cVQDyEN7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQIDAQABMA0GCSqG \
  SIb3DQEBCwUAA4IBAQBnYFK0eYHy+hVf2D58usj39lhL5znb/q9G35GBd/XsWfCE \
  wHuLOSZSUmG71bZtrOcx0ptle9bp2kKl4HlSTTfbtpuG5onSa3swRNhtKtUy5NH9 \
```

```
    W/FLViKWfoPS3kwoEpC1XqKY6l7evoTCtS+kTQRSrCe4vbNprCAZRxz6z1nEeCgu \
    NMk38yTRvx8ihZpVOuU+Ih+dOtVe/ex5IAPYxlQsvtfhsUZqc7IyCcy72WHnRHlU \
    fn3pJm0S5270+Yls3Iv6h3oBAP19i906UjiUTNH3g0xMW+V4uLxgyckt4wD4Mlyv \
    jnaQ7Z3sR6EsXMocAbXHIAJhwKdtU/fLgdwL5vtx
```

```
{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "proof": "jws",
      "key": {
        "cert": "MIIC6jCCAdKgAwIBAgIGAXjw74xPMA0GCSqGSIb3DQEBCwUAMD\
    YxNDAyBgNVBAMMK05JWU15QmpzRGp5QkM5UDUzN0Q2SVR6a3BEOE50UmppOXlhcEV\
    6QzY2bVEwHhcNMjEwNDIwMjAxODU0WhcNMjIwMjE0MjAxODU0WjA2MTQwMgYDVQQD\
    DCtOSVlNeUJqc0RqeUJDOVA1MzdENklUemtwRDhOdFJqaTl5YXBFekM2Nm1RMIIBI\
    jANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhYOJ+XOKISdMMShn/G4W9m20mT\
    0VWtQBsmBBkI2cmRt4Ai8BfYdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8\
    KowlyVy8IkZ8NMwSrcUIBZGYXjHpwjzvfGvXH/5KJlnR3/uRUp4Z4Ujk2bCaKegDn\
    11V2vxE41hqaPUnhRZxe0jRETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDad\
    z8BkPo+uv4BC0bunS0K3bA/3UgVp7zBlQFoFnLTO2uWp/muLEWGl67gBq9MO3brKX\
    fGhi3kOzywzwPTuq+cVQDyEN7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQIDAQABMA0\
    GCSqGSIb3DQEBCwUAA4IBAQBnYFK0eYHy+hVf2D58usj39lhL5znb/q9G35GBd/Xs\
    WfCEwHuLOSZSUmG71bZtrOcx0ptle9bp2kKl4HlSTTfbtpuG5onSa3swRNhtKtUy5\
    NH9W/FLViKWfoPS3kwoEpC1XqKY6l7evoTCtS+kTQRSrCe4vbNprCAZRxz6z1nEeC\
    guNMk38yTRvx8ihZpVOuU+Ih+dOtVe/ex5IAPYxlQsvtfhsUZqc7IyCcy72WHnRHl\
    Ufn3pJm0S5270+Yls3Iv6h3oBAP19i906UjiUTNH3g0xMW+V4uLxgyckt4wD4Mlyv\
    jnaQ7Z3sR6EsXMocAbXHIAJhwKdtU/fLgdwL5vtx"
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    },
    "subject": {
        "formats": ["iss_sub", "opaque"]
    }
```

```
}
```

The verifier compares the TLS client certificate presented during mutual TLS negotiation to the expected key of the signer. Since the TLS connection covers the entire message, there are no additional requirements to check.

Note that in many instances, the verifier will not do a full certificate chain validation of the presented TLS client certificate, as the means of trust for this certificate could be in something other than a PKI system, such as a static registration or trust-on-first-use.

[[See issue #110 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/110)]]

7.3.3. Detached JWS

This method is indicated by "jwsd" in the "proof" field. A JWS [RFC7515] object is created as follows:

To protect the request, the JOSE header of the signature contains the following parameters:

kid (string) The key identifier. RECOMMENDED. If the key is presented in JWK format, this MUST be the value of the "kid" field of the key.

alg (string) The algorithm used to sign the request. REQUIRED. MUST be appropriate to the key presented. If the key is presented as a JWK, this MUST be equal to the "alg" parameter of the key. MUST NOT be "none".

typ (string) The type header, value "gnap-binding+jwsd". REQUIRED

htm (string) The HTTP Method used to make this request, as an uppercase ASCII string. REQUIRED

uri (string) The HTTP URI used for this request, including all path and query components and no fragment component. REQUIRED

created (integer) A timestamp of when the signature was created, in integer seconds since UNIX Epoch

ath (string) When a request is bound to an access token, the access

token hash value. The value MUST be the result of Base64url encoding (with no padding) the SHA-256 digest of the ASCII encoding of the associated access token's value. REQUIRED if the request protects an access token.

If the HTTP request has a message body, such as an HTTP POST or PUT method, the payload of the JWS object is the Base64url encoding (without padding) of the SHA256 digest of the bytes of the body. If the request being made does not have a message body, such as an HTTP GET, OPTIONS, or DELETE method, the JWS signature is calculated over an empty payload.

The client instance presents the signed object in compact form [RFC7515] in the Detached-JWS HTTP Header field.

In this example, the JOSE Header contains the following parameters:

```
{
    "alg": "RS256",
    "kid": "gnap-rsa",
    "uri": "https://server.example.com/gnap",
    "htm": "POST",
    "typ": "gnap-binding+jwsd",
    "created": 1618884475
}
```

The request body is the following JSON object:

```
{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "proof": "jwsd",
      "key": {
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "RS256",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    }
}
```

This is hashed to the following Base64 encoded value:

```
PGiVuOZUcN1tRtUS6tx2b4cBgw9mPgXG3IPB3wY7ctc
```

This leads to the following full HTTP request message:

```
POST /gnap HTTP/1.1
Host: server.example.com
Content-Type: application/json
Content-Length: 983
Detached-JWS: eyJhbGciOiJSUzI1NiIsImNyZWF0ZWQiOjE2MTg4ODQ0NzUsImh0b\
  SI6IlBPU1QiLCJraWQiOiJnbmFwLXJzYSIsInR5cCI6ImduYXAtYmluZGluZytqd3\
  NkIiwidXJpIjoiaHR0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20vZ25hcCJ9.PGiVuO\
  ZUcN1tRtUS6tx2b4cBgw9mPgXG3IPB3wY7ctc.fUq-SV-A1iFN2MwCRW_yolVtT2_\
  TZA2h5YeXUoi5F2Q2iToC0Tc4drYFOSHIX68knd68RUA7yHqCVP-ZQEd6aL32H69e\
  9zuMiw6O_s4TBKB3vDOvwrhYtDH6fX2hP70cQoO-47OwbqP-ifkrvI3hVgMX9TfjV\
  eKNwnhoNnw3vbu7SNKeqJEbbwZfpESaGepS52xNBlDNMYBQQXxM9OqKJaXffzLFEl\
  -Xe0UnfolVtBraz3aPrPy1C6a4uT7wLda3PaTOVtgysxzii3oJWpuz0WP5kRujzDF\
  wX_EOzW0jsjCSkL-PXaKSpZgEjNjKDMg9irSxUISt1C1T6q3SzRgfuQ


{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "proof": "jwsd",
      "key": {
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "RS256",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
```

```
        },
      }
}
```

When the verifier receives the Detached-JWS header, it MUST parse and validate the JWS object. The signature MUST be validated against the expected key of the signer. All required fields MUST be present and their values MUST be valid. If the HTTP message request contains a body, the verifier MUST calculate the hash of body just as the signer does, with no normalization or transformation of the request.

7.3.4. Attached JWS

This method is indicated by "jws" in the "proof" field. A JWS [RFC7515] object is created as follows:

The JOSE header MUST contain the "kid" parameter of the key bound to this client instance for this request. The "alg" parameter MUST be set to a value appropriate for the key identified by kid and MUST NOT be "none".

To protect the request, the JWS header MUST contain the following additional parameters.

typ (string) The type header, value "gnap-binding+jws".

htm (string) The HTTP Method used to make this request, as an uppercase ASCII string.

uri (string) The HTTP URI used for this request, including all path and query components and no fragment component.

created (integer) A timestamp of when the signature was created, in integer seconds since UNIX Epoch

ath (string) When a request is bound to an access token, the access token hash value. The value MUST be the result of Base64url encoding (with no padding) the SHA-256 digest of the ASCII encoding of the associated access token’s value.

If the HTTP request has a message body, such as an HTTP POST or PUT method, the payload of the JWS object is the JSON serialized body of the request, and the object is signed according to JWS and serialized into compact form [RFC7515]. The client instance presents the JWS as the body of the request along with a content type of "application/jose". The AS MUST extract the payload of the JWS and treat it as the request body for further processing.

If the request being made does not have a message body, such as an HTTP GET, OPTIONS, or DELETE method, the JWS signature is calculated over an empty payload and passed in the "Detached-JWS" header as described in Section 7.3.3.

In this example, the JOSE header contains the following parameters:

```
{
    "alg": "RS256",
    "kid": "gnap-rsa",
    "uri": "https://server.example.com/gnap",
    "htm": "POST",
    "typ": "gnap-binding+jwsd",
    "created": 1618884475
}
```

The request body, used as the JWS Payload, is the following JSON object:

```
{
    "access_token": {
        "access": [
            "dolphin-metadata"
        ]
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.foo/callback",
            "nonce": "VJLO6A4CAYLBXHTR0KRO"
        }
    },
    "client": {
      "proof": "jws",
      "key": {
        "jwk": {
            "kid": "gnap-rsa",
            "kty": "RSA",
            "e": "AQAB",
            "alg": "RS256",
            "n": "hYOJ-XOKISdMMShn_G4W9m20mT0VWtQBsmBBkI2cmRt4Ai8Bf\
  YdHsFzAtYKOjpBR1RpKpJmVKxIGNy0g6Z3ad2XYsh8KowlyVy8IkZ8NMwSrcUIBZG\
  YXjHpwjzvfGvXH_5KJlnR3_uRUp4Z4Ujk2bCaKegDn11V2vxE41hqaPUnhRZxe0jR\
  ETddzsE3mu1SK8dTCROjwUl14mUNo8iTrTm4n0qDadz8BkPo-uv4BC0bunS0K3bA_\
  3UgVp7zBlQFoFnLTO2uWp_muLEWGl67gBq9MO3brKXfGhi3kOzywzwPTuq-cVQDyE\
  N7aL0SxCb3Hc4IdqDaMg8qHUyObpPitDQ"
        }
      }
      "display": {
        "name": "My Client Display Name",
        "uri": "https://client.foo/"
      },
    },
    "subject": {
        "formats": ["iss_sub", "opaque"]
    }
}
```

This leads to the following full HTTP request message:

```
POST /gnap HTTP/1.1
Host: server.example.com
Content-Type: application/jose
Content-Length: 1047

eyJhbGciOiJSUzI1NiIsImNyZWF0ZWQiOjE2MTg4ODQ0NzUsImh0bSI6IlBPU1QiLCJ\
raWQiOiJnbmFwLXJzYSIsInR5cCI6ImduYXAtYmluZGluZytqd3NkIiwidXJpIjoiaH\
R0cHM6Ly9zZXJ2ZXIuZXhhbXBsZS5jb20vZ25hcCJ9.CnsKICAgICJhY2Nlc3NfdG9r\
ZW4iOiB7CiAgICAgICAgImFjY2VzcyI6IFsKICAgICAgICAgICAgImRvbHBoaW4tbWV\
0YWRhdGEiCiAgICAgICAgXQogICAgfSwKICAgICJpbnRlcmFjdCI6IHsKICAgICAgIC\
Aic3RhcnQiOiBbInJlZGlyZWN0Il0sCiAgICAgICAgImZpbmlzaCI6IHsKICAgICAgI\
CAgICAgIm1ldGhvZCI6ICJyZWRpcmVjdCIsCiAgICAgICAgICAgICJ1cmkiOiAiaHR0\
cHM6Ly9jbGllbnQuZm9vL2NhbGxiYWNrIiwKICAgICAgICAgICAgIm5vbmNlIjogIlZ\
KTE82QTRDQVlMQlhIVFIwS1JPIgogICAgICAgIH0KICAgICAgICAgIH0sCiAgICAiY2xpZW50Ij\
ogewogICAgICAicHJvb2YiOiAiandzIiwKICAgICAgImtleSI6IHsKICAgICAgICAia\
ndrIjogewogICAgICAgICAgICAia2lkIjogImduYXAtcnNhIiwKICAgICAgICAgICAg\
Imt0eSI6ICJSU0EiLAogICAgICAgICAgICAiZSI6ICJBUUFCIiwKICAgICAgICAgICA\
gImFsZyI6ICJSUzI1NiIsCiAgICAgICAgICAgICJuIjogImhZT0otWE9LSVNkTU1TaG\
5fRzRXOW0yMG1UMFZXdFFCc21CQmtJMmNtUnQ0QWk4QmZZZEhzRnpBdFlLT2pwQlIxU\
nBLcEptVkt4SUdOeTBnNlozYWQyWFlzaDhLb3dseVZ5OElrWjhOTXdTcmNVSUJaR1lY\
akhwd2p6dmZHdlhIXzVLSmxuUjNfdVJVcDRaNFVqazJiQ2FLZWdEbjExVjJ2eEU0MWh\
xYVBVbmhSWnhlMGpSRVRkZHpzRTNtdTFTSzhkVENST2p3VWwxNG1VTm84aVRyVG00bj\
BxRGFkejhCa1BvLXV2NEJDMGJ1blMwSzNiQV8zVWdWcDd6QmxRRm9GbkxUTzJ1V3Bfb\
XVMRVdHbDY3Z0JxOU1PM2JyS1hmR2hpM2tPenl3endQVHVxLWNWUUR5RU43YUwwU3hD\
YjNIYzRJZHFEYU1nOHFIVXlPYnBQaXREUSIKICAgICAgICB9CiAgICAgICAgIH0KICAgICA\
gImRpc3BsYXkiOiB7CiAgICAgICAgICAgIm5hbWUiOiAiTXkgQ2xpZW50IERpc3BsYXkgTm\
FtZSIsCiAgICAgICAgICAgInVyaSI6ICJodHRwczovL2NsaWVudC5mb28vIgogICAgICB9L\
AogICAgfSwKICAgICJzdWJqZWN0IjogewogICAgICAgICJmb3JtYXRzIjogWyJpc3Nf\
c3ViIiwgIm9wYXF1ZSJdCiAgICB9Cn0K.MwNoVMQp5hVxI0mCs9LlOUdFtkDXaA1_eT\
vOXq7DOGrtDKH7q4vP2xUq3fH2jRAZqnobo0WdPP3eM3NH5QUjW8pa6_QpwdIWkK7r-\
u_52puE0lPBp7J4U2w4l9gIbg8iknsmWmXeY5F6wiGT8ptfuEYGgmloAJd9LIeNvD3U\
LW2h2dz1Pn2eDnbyvgB0Ugae0BoZB4f69fKWj8Z9wvTIjk1LZJN1PcL7_zT8Lrlic9a\
PyzT7Q9ovkd1s-4whE7TrnGUzFc5mgWUn_gsOpsP5mIIljoEEv-FqOW2RyNYulOZl0Q\
8EnnDHV_vPzrHlUarbGg4YffgtwkQhdK72-JOxYQ
```

[[See issue #109 (https://github.com/ietf-wg-gnap/gnap-core-protocol/issues/109)]]

When the verifier receives an attached JWS request, it MUST parse and validate the JWS object. The signature MUST be validated against the expected key of the signer. All required fields MUST be present and their values MUST be valid. If the HTTP message request contains a body, the verifier MUST decode the payload of the JWS object and treat this as the HTTP message body.

8. Resource Access Rights

GNAP provides a rich structure for describing the protected resources hosted by RSs and accessed by client software. This structure is used when the client instance requests an access token (Section 2.1) and when an access token is returned (Section 3.2).

The root of this structure is a JSON array. The elements of the JSON array represent rights of access that are associated with the the access token. The resulting access is the union of all elements within the array.

The access associated with the access token is described using objects that each contain multiple dimensions of access. Each object contains a REQUIRED "type" property that determines the type of API that the token is used for.

type (string) The type of resource request as a string. This field MAY define which other fields are allowed in the request object. This field is REQUIRED.

The value of the "type" field is under the control of the AS. This field MUST be compared using an exact byte match of the string value against known types by the AS. The AS MUST ensure that there is no collision between different authorization data types that it supports. The AS MUST NOT do any collation or normalization of data types during comparison. It is RECOMMENDED that designers of general-purpose APIs use a URI for this field to avoid collisions between multiple API types protected by a single AS.

While it is expected that many APIs will have their own properties, a set of common properties are defined here. Specific API implementations SHOULD NOT re-use these fields with different semantics or syntax. The available values for these properties are determined by the API being protected at the RS.

actions (array of strings) The types of actions the client instance will take at the RS as an array of strings. For example, a client instance asking for a combination of "read" and "write" access.

locations (array of strings) The location of the RS as an array of strings. These strings are typically URIs identifying the location of the RS.

datatypes (array of strings) The kinds of data available to the client instance at the RS's API as an array of strings. For example, a client instance asking for access to raw "image" data and "metadata" at a photograph API.

identifier (string) A string identifier indicating a specific resource at the RS. For example, a patient identifier for a medical API or a bank account number for a financial API.

privileges (array of strings) The types or levels of privilege being requested at the resource. For example, a client instance asking for administrative level access, or access when the resource owner is no longer online.

The following non-normative example is describing three kinds of access (read, write, delete) to each of two different locations and two different data types (metadata, images) for a single access token using the fictitious "photo-api" type definition.

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read",
            "write",
            "delete"
        ],
        "locations": [
            "https://server.example.net/",
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata",
            "images"
        ]
    }
]
```

The access requested for a given object when using these fields is the cross-product of all fields of the object. That is to say, the object represents a request for all "actions" listed to be used at all "locations" listed for all possible "datatypes" listed within the object. Assuming the request above was granted, the client instance could assume that it would be able to do a "read" action against the "images" on the first server as well as a "delete" action on the "metadata" of the second server, or any other combination of these fields, using the same access token.

To request a different combination of access, such as requesting one of the possible "actions" against one of the possible "locations" and a different choice of possible "actions" against a different one of the possible "locations", the client instance can include multiple separate objects in the "resources" array. The following non-

normative example uses the same fictitious "photo-api" type definition to request a single access token with more specifically targeted access rights by using two discrete objects within the request.

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read"
        ],
        "locations": [
            "https://server.example.net/"
        ],
        "datatypes": [
            "images"
        ]
    },
    {
        "type": "photo-api",
        "actions": [
            "write",
            "delete"
        ],
        "locations": [
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata"
        ]
    }
]
```

The access requested here is for "read" access to "images" on one server while simultaneously requesting "write" and "delete" access for "metadata" on a different server, but importantly without requesting "write" or "delete" access to "images" on the first server.

It is anticipated that API designers will use a combination of common fields defined in this specification as well as fields specific to the API itself. The following non-normative example shows the use of both common and API-specific fields as part of two different fictitious API "type" values. The first access request includes the "actions", "locations", and "datatypes" fields specified here as well as the API-specific "geolocation" field. The second access request includes the "actions" and "identifier" fields specified here as well as the API-specific "currency" field.

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read",
            "write"
        ],
        "locations": [
            "https://server.example.net/",
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata",
            "images"
        ],
        "geolocation": [
            { lat: -32.364, lng: 153.207 },
            { lat: -35.364, lng: 158.207 }
        ]
    },
    {
        "type": "financial-transaction",
        "actions": [
            "withdraw"
        ],
        "identifier": "account-14-32-32-3",
        "currency": "USD"
    }
]
```

If this request is approved, the resulting access token (Section 3.2.1)'s access rights will be the union of the requested types of access for each of the two APIs, just as above.

8.1. Requesting Resources By Reference

Instead of sending an object describing the requested resource (Section 8), access rights MAY be communicated as a string known to the AS or RS representing the access being requested. Each string SHOULD correspond to a specific expanded object representation at the AS.

```
"access": [
    "read", "dolphin-metadata", "some other thing"
]
```

This value is opaque to the client instance and MAY be any valid JSON string, and therefore could include spaces, unicode characters, and properly escaped string sequences. However, in some situations the value is intended to be seen and understood by the client software's developer. In such cases, the API designer choosing any such human-readable strings SHOULD take steps to ensure the string values are not easily confused by a developer, such as by limiting the strings to easily disambiguated characters.

This functionality is similar in practice to OAuth 2.0's "scope" parameter [RFC6749], where a single string represents the set of access rights requested by the client instance. As such, the reference string could contain any valid OAuth 2.0 scope value as in Appendix D.5. Note that the reference string here is not bound to the same character restrictions as in OAuth 2.0's "scope" definition.

A single "access" array MAY include both object-type and string-type resource items. In this non-normative example, the client instance is requesting access to a "photo-api" and "financial-transaction" API type as well as the reference values of "read", "dolphin-metadata", and "some other thing".

```
"access": [
    {
        "type": "photo-api",
        "actions": [
            "read",
            "write",
            "delete"
        ],
        "locations": [
            "https://server.example.net/",
            "https://resource.local/other"
        ],
        "datatypes": [
            "metadata",
            "images"
        ]
    },
    "read",
    "dolphin-metadata",
    {
        "type": "financial-transaction",
        "actions": [
            "withdraw"
        ],
        "identifier": "account-14-32-32-3",
        "currency": "USD"
    },
    "some other thing"
]
```

The requested access is the union of all elements of the array, including both objects and reference strings.

9. Discovery

By design, the protocol minimizes the need for any pre-flight discovery. To begin a request, the client instance only needs to know the endpoint of the AS and which keys it will use to sign the request. Everything else can be negotiated dynamically in the course of the protocol.

However, the AS can have limits on its allowed functionality. If the client instance wants to optimize its calls to the AS before making a request, it MAY send an HTTP OPTIONS request to the grant request endpoint to retrieve the server's discovery information. The AS MUST respond with a JSON document containing the following information:

grant_request_endpoint (string) REQUIRED. The location of the AS's

grant request endpoint. The location MUST be a URL [RFC3986] with a scheme component that MUST be https, a host component, and optionally, port, path and query components and no fragment components. This URL MUST match the URL the client instance used to make the discovery request.

interaction_start_modes_supported (array of strings) OPTIONAL. A list of the AS's interaction start methods. The values of this list correspond to the possible values for the interaction start section (Section 2.5.1) of the request.

interaction_finish_methods_supported (array of strings) OPTIONAL. A list of the AS's interaction finish methods. The values of this list correspond to the possible values for the method element of the interaction finish section (Section 2.5.2) of the request.

key_proofs_supported (array of strings) OPTIONAL. A list of the AS's supported key proofing mechanisms. The values of this list correspond to possible values of the "proof" field of the key section (Section 7.1) of the request.

subject_formats_supported (array of strings) OPTIONAL. A list of the AS's supported subject identifier types. The values of this list correspond to possible values of the subject identifier section (Section 2.2) of the request.

assertions_supported (array of strings) OPTIONAL. A list of the AS's supported assertion formats. The values of this list correspond to possible values of the subject assertion section (Section 2.2) of the request.

The information returned from this method is for optimization purposes only. The AS MAY deny any request, or any portion of a request, even if it lists a capability as supported. For example, a given client instance can be registered with the "mtls" key proofing mechanism, but the AS also returns other proofing methods, then the AS will deny a request from that client instance using a different proofing mechanism.

9.1. RS-first Method of AS Discovery

If the client instance calls an RS without an access token, or with an invalid access token, the RS MAY respond to the client instance with an authentication header indicating that GNAP needs to be used to access the resource. The address of the GNAP endpoint MUST be sent in the "as_uri" parameter. The RS MAY additionally return a resource reference that the client instance MAY use in its access token request. This resource reference MUST be sufficient for at

least the action the client instance was attempting to take at the RS and MAY be more powerful. The means for the RS to determine the resource reference are out of scope of this specification, but some dynamic methods are discussed in [I-D.draft-ietf-gnap-resource-servers]. The content of the resource handle is opaque to the client instance.

```
WWW-Authenticate: \
  GNAP as_uri=https://server.example/tx,access=FWWIKYBQ6U56NL1
```

The client instance then makes a request to the "as_uri" as described in Section 2, with the value of "access" as one of the members of the "access" array in the "access_token" portion of the request. The client instance MAY request additional resources and other information. The client instance MAY request multiple access tokens.

In this non-normative example, the client instance is requesting a single access token using the resource reference "FWWIKYBQ6U56NL1" received from the RS in addition to the "dolphin-metadata" resource reference that the client instance has been configured with out of band.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
    "access_token": {
        "access": [
            "FWWIKYBQ6U56NL1",
            "dolphin-metadata"
        ]
    },
    "client": "KHRS6X63AJ7C7C4AZ9AO"
}
```

If issued, the resulting access token would contain sufficient access to be used at both referenced resources.

10. Acknowledgements

The editors would like to thank the feedback of the following individuals for their reviews, implementations, and contributions: Aeke Axeland, Aaron Parecki, Adam Omar Oueidat, Annabelle Backman, Dick Hardt, Dmitri Zagidulin, Dmitry Barinov, Fabien Imbault, Francis Pouatcha, George Fletcher, Haardik Haardik, Hamid Massaoud, Jacky Yuan, Joseph Heenan, Justin Richer, Kathleen Moriarty, Mike Jones, Mike Varley, Nat Sakimura, Takahiko Kawasaki, Takahiro Tsuchiya.

The editors would also like to thank the GNAP working group design team of Kathleen Moriarty, Fabien Imbault, Dick Hardt, Mike Jones, and Justin Richer, who incorporated elements from the XAuth and XYZ proposals to create the first version of this document.

In addition, the editors would like to thank Aaron Parecki and Mike Jones for insights into how to integrate identity and authentication systems into the core protocol, and Justin Richer and Dick Hardt for the use cases, diagrams, and insights provided in the XYZ and XAuth proposals that have been incorporated here. The editors would like to especially thank Mike Varley and the team at SecureKey for feedback and development of early versions of the XYZ protocol that fed into this standards work.

11. IANA Considerations

[[TBD: There are a lot of items in the document that are expandable through the use of value registries.]]

12. Security Considerations

[[TBD: There are a lot of security considerations to add.]]

All requests have to be over TLS or equivalent as per [BCP195]. Many handles act as shared secrets, though they can be combined with a requirement to provide proof of a key as well.

13. Privacy Considerations

[[TBD: There are a lot of privacy considerations to add.]]

Handles are passed between parties and therefore should not contain any private data.

When user information is passed to the client instance, the AS needs to make sure that it has the permission to do so.

14. Normative References

[BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", May 2015, <https://www.rfc-editor.org/info/bcp195>.

[I-D.draft-ietf-gnap-resource-servers] Richer, J., Parecki, A., and F. Imbault, "Grant Negotiation and Authorization Protocol Resource Server Connections", Work in Progress, Internet-Draft, draft-ietf-gnap-resource-servers-00, 28 April 2021, <https://www.ietf.org/archive/id/draft-ietf-gnap-resource-servers-00.txt>.

[I-D.ietf-httpbis-message-signatures] Backman, A., Richer, J., and M. Sporny, "Signing HTTP Messages", Work in Progress, Internet-Draft, draft-ietf-httpbis-message-signatures-05, 8 June 2021, <https://www.ietf.org/archive/id/draft-ietf-httpbis-message-signatures-05.txt>.

[I-D.ietf-oauth-rar] Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", Work in Progress, Internet-Draft, draft-ietf-oauth-rar-05, 15 May 2021, <https://www.ietf.org/archive/id/draft-ietf-oauth-rar-05.txt>.

[I-D.ietf-oauth-signed-http-request] Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", Work in Progress, Internet-Draft, draft-ietf-oauth-signed-http-request-03, 8 August 2016, <https://www.ietf.org/archive/id/draft-ietf-oauth-signed-http-request-03.txt>.

[I-D.ietf-secevent-subject-identifiers] Backman, A. and M. Scurtescu, "Subject Identifiers for Security Event Tokens", Work in Progress, Internet-Draft, draft-ietf-secevent-subject-identifiers-08, 24 May 2021, <https://www.ietf.org/archive/id/draft-ietf-secevent-subject-identifiers-08.txt>.

[OIDC] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1", November 2014, <https://openiD.net/specs/openiD-connect-core-1_0.html>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3230] Mogul, J. and A. Van Hoff, "Instance Digests in HTTP", RFC 3230, DOI 10.17487/RFC3230, January 2002, <https://www.rfc-editor.org/info/rfc3230>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", BCP 47, RFC 5646, DOI 10.17487/RFC5646, September 2009, <https://www.rfc-editor.org/info/rfc5646>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/info/rfc6750>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching", RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://www.rfc-editor.org/info/rfc7234>.

[RFC7468] Josefsson, S. and S. Leonard, "Textual Encodings of PKIX, PKCS, and CMS Structures", RFC 7468, DOI 10.17487/RFC7468, April 2015, <https://www.rfc-editor.org/info/rfc7468>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", RFC 8705, DOI 10.17487/RFC8705, February 2020, <https://www.rfc-editor.org/info/rfc8705>.

[RFC8792] Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <https://www.rfc-editor.org/info/rfc8792>.

Appendix A. Document History

* -06

  - Removed "capabilities" and "existing_grant" protocol fields.

  - Removed separate "instance_id" field.

  - Split "interaction_methods_supported" into "interaction_start_modes_supported" and "interaction_finish_methods_supported".

  - Added AS endpoint to hash calculation to fix mix-up attack.

  - Added "privileges" field to resource access request object.

  - Moved client-facing RS response back from GNAP-RS document.

  - Removed oauthpop key binding.

  - Removed dpop key binding.

  - Added example DID identifier.

  - Changed token response booleans to flag structure to match request.

  - Updated signature examples to use HTTP Message Signatures.

* -05

  - Changed "interaction_methods" to "interaction_methods_supported".

  - Changed "key_proofs" to "key_proofs_supported".

  - Changed "assertions" to "assertions_supported".

  - Updated discovery and field names for subject formats.

  - Add an appendix to provide protocol rationale, compared to OAuth2.

  - Updated subject information definition.

  - Refactored the RS-centric components into a new document.

  - Updated cryptographic proof of possession methods to match current reference syntax.

  - Updated proofing language to use "signer" and "verifier" generically.

  - Updated cryptographic proof of possession examples.

  - Editorial cleanup and fixes.

  - Diagram cleanup and fixes.

* -04

  - Updated terminology.

  - Refactored key presentation and binding.

  - Refactored "interact" request to group start and end modes.

  - Changed access token request and response syntax.

  - Changed DPoP digest field to ’htd’ to match proposed FAPI profile.

  - Include the access token hash in the DPoP message.

  - Removed closed issue links.

  - Removed function to read state of grant request by client.

  - Closed issues related to reading and updating access tokens.

* -03

  - Changed "resource client" terminology to separate "client instance" and "client software".

  - Removed OpenID Connect "claims" parameter.

  - Dropped "short URI" redirect.

  - Access token is mandatory for continuation.

  - Removed closed issue links.

  - Editorial fixes.

* -02

  - Moved all "editor's note" items to GitHub Issues.

  - Added JSON types to fields.

  - Changed "GNAP Protocol" to "GNAP".

  - Editorial fixes.

* -01

  - "updated_at" subject info timestamp now in ISO 8601 string format.

  - Editorial fixes.

  - Added Aaron and Fabien as document authors.

* -00

  - Initial working group draft.

Appendix B. Compared to OAuth 2.0

GNAP's protocol design differs from OAuth 2.0's in several fundamental ways:

1. *Consent and authorization flexibility:*

   OAuth 2.0 generally assumes the user has access to the a web browser. The type of interaction available is fixed by the grant type, and the most common interactive grant types start in the

browser. OAuth 2.0 assumes that the user using the client software is the same user that will interact with the AS to approve access.

GNAP allows various patterns to manage authorizations and consents required to fulfill this requested delegation, including information sent by the client instance, information supplied by external parties, and information gathered through the interaction process. GNAP allows a client instance to list different ways that it can start and finish an interaction, and these can be mixed together as needed for different use cases. GNAP interactions can use a browser, but don’t have to. Methods can use inter-application messaging protocols, out-of-band data transfer, or anything else. GNAP allows extensions to define new ways to start and finish an interaction, as new methods and platforms are expected to become available over time. GNAP is designed to allow the end-user and the resource owner to be two different people, but still works in the optimized case of them being the same party.

2. *Intent registration and inline negotiation:*

   OAuth 2.0 uses different "grant types" that start at different endpoints for different purposes. Many of these require discovery of several interrelated parameters.

   GNAP requests all start with the same type of request to the same endpoint at the AS. Next steps are negotiated between the client instance and AS based on software capabilities, policies surrounding requested access, and the overall context of the ongoing request. GNAP defines a continuation API that allows the client instance and AS to request and send additional information from each other over multiple steps. This continuation API uses the same access token protection that other GNAP-protected APIs use. GNAP allows discovery to optimize the requests but it isn’t required thanks to the negotiation capabilities.

3. *Client instances:*

   OAuth 2.0 requires all clients to be registered at the AS and to use a client_id known to the AS as part of the protocol. This client_id is generally assumed to be assigned by a trusted authority during a registration process, and OAuth places a lot of trust on the client_id as a result. Dynamic registration allows different classes of clients to get a client_id at runtime, even if they only ever use it for one request.

GNAP allows the client instance to present an unknown key to the AS and use that key to protect the ongoing request. GNAP's client instance identifier mechanism allows for pre-registered clients and dynamically registered clients to exist as an optimized case without requiring the identifier as part of the protocol at all times.

4. *Expanded delegation:*

   OAuth 2.0 defines the "scope" parameter for controlling access to APIs. This parameter has been coopted to mean a number of different things in different protocols, including flags for turning special behavior on and off, including the return of data apart from the access token. The "resource" parameter and RAR extensions (as defined in [I-D.ietf-oauth-rar]) expand on the "scope" concept in similar but different ways.

   GNAP defines a rich structure for requesting access, with string references as an optimization. GNAP defines methods for requesting directly-returned user information, separate from API access. This information includes identifiers for the current user and structured assertions. The core GNAP protocol makes no assumptions or demands on the format or contents of the access token, but the RS extension allows a negotiation of token formats between the AS and RS.

5. *Cryptography-based security:*

   OAuth 2.0 uses shared bearer secrets, including the client_secret and access token, and advanced authentication and sender constraint have been built on after the fact in inconsistent ways.

   In GNAP, all communication between the client instance and AS is bound to a key held by the client instance. GNAP uses the same cryptographic mechanisms for both authenticating the client (to the AS) and binding the access token (to the RS and the AS). GNAP allows extensions to define new cryptographic protection mechanisms, as new methods are expected to become available over time. GNAP does not have a notion of "public clients" because key information can always be sent and used dynamically.

6. *Privacy and usable security:*

   OAuth 2.0's deployment model assumes a strong binding between the AS and the RS.

GNAP is designed to be interoperable with decentralized identity standards and to provide a human-centric authorization layer. In addition to the core protocol, GNAP that supports various patterns of communication between RSs and ASs through extensions. GNAP tries to limit the odds of a consolidation to just a handful of super-popular AS services.

Appendix C. Component Data Models

While different implementations of this protocol will have different realizations of all the components and artifacts enumerated here, the nature of the protocol implies some common structures and elements for certain components. This appendix seeks to enumerate those common elements.

TBD: Client has keys, allowed requested resources, identifier(s), allowed requested subjects, allowed

TBD: AS has "grant endpoint", interaction endpoints, store of trusted client keys, policies

TBD: Token has RO, user, client, resource list, RS list,

Appendix D. Example Protocol Flows

The protocol defined in this specification provides a number of features that can be combined to solve many different kinds of authentication scenarios. This section seeks to show examples of how the protocol would be applied for different situations.

Some longer fields, particularly cryptographic information, have been truncated for display purposes in these examples.

D.1. Redirect-Based User Interaction

In this scenario, the user is the RO and has access to a web browser, and the client instance can take front-channel callbacks on the same device as the user. This combination is analogous to the OAuth 2.0 Authorization Code grant type.

The client instance initiates the request to the AS. Here the client instance identifies itself using its public key.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
    "access_token": {
        "access": [
            {
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            }
        ],
    },
    "client": {
      "key": {
        "proof": "httpsig",
        "jwk": {
            "kty": "RSA",
            "e": "AQAB",
            "kid": "xyz-1",
            "alg": "RS256",
            "n": "kOB5rR4Jv0GMeLaY6_It_r3ORwdf8ci_JtffXyaSx8..."
        }
      }
    },
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return/123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    }
}
```

The AS processes the request and determines that the RO needs to interact. The AS returns the following response giving the client instance the information it needs to connect. The AS has also indicated to the client instance that it can use the given instance identifier to identify itself in future requests (Section 2.3.1).

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "interact": {
       "redirect":
         "https://server.example.com/interact/4CF492MLVMSW9MKM",
       "push": "MBDOFXG4Y5CVJCX821LH"
    }
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue"
    },
    "instance_id": "7C7C4AZ9KHRS6X63AJAO"
}
```

The client instance saves the response and redirects the user to the interaction_url by sending the following HTTP message to the user's browser.

```
HTTP 302 Found
Location: https://server.example.com/interact/4CF492MLVMSW9MKM
```

The user's browser fetches the AS's interaction URL. The user logs in, is identified as the RO for the resource being requested, and approves the request. Since the AS has a callback parameter, the AS generates the interaction reference, calculates the hash, and redirects the user back to the client instance with these additional values added as query parameters.

```
HTTP 302 Found
Location: https://client.example.net/return/123455\
  ?hash=p28jsq0Y2KK3WS__a42tavNC64ldGTBroywsWxT4md_jZQ1R2\
    HZT8BOWYHcLmObM7XHPAdJzTZMtKBsaraJ64A\
  &interact_ref=4IFWWIKYBC2PQ6U56NL1
```

The client instance receives this request from the user's browser. The client instance ensures that this is the same user that was sent out by validating session information and retrieves the stored

pending request. The client instance uses the values in this to validate the hash parameter. The client instance then calls the continuation URL and presents the handle and interaction reference in the request body. The client instance signs the request as above.

```
POST /continue HTTP/1.1
Host: server.example.com
Content-Type: application/json
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
    "interact_ref": "4IFWWIKYBC2PQ6U56NL1"
}
```

The AS retrieves the pending request based on the handle and issues a bearer access token and returns this to the client instance.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [{
            "actions": [
                "read",
                "write",
                "dolphin"
            ],
            "locations": [
                "https://server.example.net/",
                "https://resource.local/other"
            ],
            "datatypes": [
                "metadata",
                "images"
            ]
        }]
    },
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue"
    }
}
```

D.2. Secondary Device Interaction

In this scenario, the user does not have access to a web browser on the device and must use a secondary device to interact with the AS. The client instance can display a user code or a printable QR code. The client instance is not able to accept callbacks from the AS and needs to poll for updates while waiting for the user to authorize the request.

The client instance initiates the request to the AS.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
    "access_token": {
        "access": [
            "dolphin-metadata", "some other thing"
        ],
    },
    "client": "7C7C4AZ9KHRS6X63AJAO",
    "interact": {
        "start": ["redirect", "user_code"]
    }
}
```

The AS processes this and determines that the RO needs to interact. The AS supports both redirect URIs and user codes for interaction, so it includes both. Since there is no "callback" the AS does not include a nonce, but does include a "wait" parameter on the continuation section because it expects the client instance to poll for results.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "interact": {
        "redirect": "https://srv.ex/MXKHQ",
        "user_code": {
            "code": "A1BC-3DFF",
            "url": "https://srv.ex/device"
        }
    },
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue/VGJKPTKC50",
        "wait": 60
    }
}
```

The client instance saves the response and displays the user code visually on its screen along with the static device URL. The client instance also displays the short interaction URL as a QR code to be scanned.

If the user scans the code, they are taken to the interaction endpoint and the AS looks up the current pending request based on the incoming URL. If the user instead goes to the static page and enters the code manually, the AS looks up the current pending request based on the value of the user code. In both cases, the user logs in, is identified as the RO for the resource being requested, and approves the request. Once the request has been approved, the AS displays to the user a message to return to their device.

Meanwhile, the client instance periodically polls the AS every 60 seconds at the continuation URL. The client instance signs the request using the same key and method that it did in the first request.

```
POST /continue/VGJKPTKC50 HTTP/1.1
Host: server.example.com
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...
```

The AS retrieves the pending request based on the handle and determines that it has not yet been authorized. The AS indicates to the client instance that no access token has yet been issued but it can continue to call after another 60 second timeout.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "continue": {
        "access_token": {
            "value": "G7YQT4KQQ5TZY9SLSS5E"
        },
        "uri": "https://server.example.com/continue/ATWHO4Q1WV",
        "wait": 60
    }
}
```

Note that the continuation URL and access token have been rotated since they were used by the client instance to make this call. The client instance polls the continuation URL after a 60 second timeout using this new information.

```
POST /continue/ATWHO4Q1WV HTTP/1.1
Host: server.example.com
Authorization: GNAP G7YQT4KQQ5TZY9SLSS5E
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...
```

The AS retrieves the pending request based on the URL and access token, determines that it has been approved, and issues an access token for the client to use at the RS.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [
            "dolphin-metadata", "some other thing"
        ]
    }
}
```

D.3. No User Involvement

In this scenario, the client instance is requesting access on its own behalf, with no user to interact with.

The client instance creates a request to the AS, identifying itself with its public key and using MTLS to make the request.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json

{
    "access_token": {
        "access": [
            "backend service", "nightly-routine-3"
        ],
    },
    "client": {
      "key": {
        "proof": "mtls",
        "cert#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o9ltc05O89jdN-dg2"
      }
    }
}
```

The AS processes this and determines that the client instance can ask for the requested resources and issues an access token.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token",
        "access": [
            "backend service", "nightly-routine-3"
        ]
    }
}
```

D.4. Asynchronous Authorization

In this scenario, the client instance is requesting on behalf of a specific RO, but has no way to interact with the user. The AS can asynchronously reach out to the RO for approval in this scenario.

The client instance starts the request at the AS by requesting a set of resources. The client instance also identifies a particular user.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
    "access_token": {
        "access": [
            {
                "type": "photo-api",
                "actions": [
                    "read",
                    "write",
                    "dolphin"
                ],
                "locations": [
                    "https://server.example.net/",
                    "https://resource.local/other"
                ],
                "datatypes": [
                    "metadata",
                    "images"
                ]
            },
            "read", "dolphin-metadata",
            {
                "type": "financial-transaction",
                "actions": [
                    "withdraw"
                ],
                "identifier": "account-14-32-32-3",
                "currency": "USD"
            },
            "some other thing"
        ],
    },
    "client": "7C7C4AZ9KHRS6X63AJAO",
    "user": {
        "sub_ids": [ {
            "format": "opaque",
            "id": "J2G8G8O4AZ"
        } ]
   }
}
```

The AS processes this and determines that the RO needs to interact. The AS determines that it can reach the identified user asynchronously and that the identified user does have the ability to approve this request. The AS indicates to the client instance that it can poll for continuation.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "continue": {
        "access_token": {
            "value": "80UPRY5NM33OMUKMKSKU"
        },
        "uri": "https://server.example.com/continue",
        "wait": 60
    }
}
```

The AS reaches out to the RO and prompts them for consent. In this example, the AS has an application that it can push notifications in to for the specified account.

Meanwhile, the client instance periodically polls the AS every 60 seconds at the continuation URL.

```
POST /continue HTTP/1.1
Host: server.example.com
Authorization: GNAP 80UPRY5NM33OMUKMKSKU
Signature-Input: sig1=...
Signature: sig1=...
```

The AS retrieves the pending request based on the handle and determines that it has not yet been authorized. The AS indicates to the client instance that no access token has yet been issued but it can continue to call after another 60 second timeout.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "continue": {
        "access_token": {
            "value": "BI9QNW6V9W3XFJK4R02D"
        },
        "uri": "https://server.example.com/continue",
        "wait": 60
    }
}
```

Note that the continuation handle has been rotated since it was used by the client instance to make this call. The client instance polls the continuation URL after a 60 second timeout using the new handle.

```
POST /continue HTTP/1.1
Host: server.example.com
Authorization: GNAP BI9QNW6V9W3XFJK4R02D
Signature-Input: sig1=...
Signature: sig1=...
```

The AS retrieves the pending request based on the handle and determines that it has been approved and it issues an access token.

```
HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store

{
    "access_token": {
        "value": "OS9M2PMHKUR64TB8N6BW7OZB8CDFONP219RP1LT0",
        "manage": "https://server.example.com/token/PRY5NM33O\
            M4TB8N6BW7OZB8CDFONP219RP1L",
        "access": [
            "dolphin-metadata", "some other thing"
        ]
    }
}
```

D.5. Applying OAuth 2.0 Scopes and Client IDs

While GNAP is not designed to be directly compatible with OAuth 2.0 [RFC6749], considerations have been made to enable the use of OAuth 2.0 concepts and constructs more smoothly within GNAP.

In this scenario, the client developer has a "client_id" and set of "scope" values from their OAuth 2.0 system and wants to apply them to the new protocol. Traditionally, the OAuth 2.0 client developer would put their "client_id" and "scope" values as parameters into a redirect request to the authorization endpoint.

```
HTTP 302 Found
Location: https://server.example.com/authorize
  ?client_id=7C7C4AZ9KHRS6X63AJAO
  &scope=read%20write%20dolphin
  &redirect_uri=https://client.example.net/return
  &response_type=code
  &state=123455
```

Now the developer wants to make an analogous request to the AS using GNAP. To do so, the client instance makes an HTTP POST and places the OAuth 2.0 values in the appropriate places.

```
POST /tx HTTP/1.1
Host: server.example.com
Content-Type: application/json
Signature-Input: sig1=...
Signature: sig1=...
Digest: sha256=...

{
    "access_token": {
        "access": [
            "read", "write", "dolphin"
        ],
        "flags": [ "bearer" ]
    },
    "client": "7C7C4AZ9KHRS6X63AJAO",
    "interact": {
        "start": ["redirect"],
        "finish": {
            "method": "redirect",
            "uri": "https://client.example.net/return?state=123455",
            "nonce": "LKLTI25DK82FX4T4QFZC"
        }
    }
}
```

The "client_id" can be used to identify the client instance's keys that it uses for authentication, the scopes represent resources that the client instance is requesting, and the "redirect_uri" and "state" value are pre-combined into a "finish" URI that can be unique per request. The client instance additionally creates a nonce to protect the callback, separate from the state parameter that it has added to its return URL.

From here, the protocol continues as above.

Appendix E. JSON Structures and Polymorphism

GNAP makes use of polymorphism within the JSON [RFC8259] structures used for the protocol. Each portion of this protocol is defined in terms of the JSON data type that its values can take, whether it's a string, object, array, boolean, or number. For some fields, different data types offer different descriptive capabilities and are used in different situations for the same field. Each data type provides a different syntax to express the same underlying semantic protocol element, which allows for optimization and simplification in many common cases.

Even though JSON is often used to describe strongly typed structures, JSON on its own is naturally polymorphic. In JSON, the named members of an object have no type associated with them, and any data type can be used as the value for any member. In practice, each member has a semantic type that needs to make sense to the parties creating and consuming the object. Within this protocol, each object member is defined in terms of its semantic content, and this semantic content might have expressions in different concrete data types for different specific purposes. Since each object member has exactly one value in JSON, each data type for an object member field is naturally mutually exclusive with other data types within a single JSON object.

For example, a resource request for a single access token is composed of an array of resource request descriptions while a request for multiple access tokens is composed of an object whose member values are all arrays. Both of these represent requests for access, but the difference in syntax allows the client instance and AS to differentiate between the two request types in the same request.

Another form of polymorphism in JSON comes from the fact that the values within JSON arrays need not all be of the same JSON data type. However, within this protocol, each element within the array needs to be of the same kind of semantic element for the collection to make sense, even when the data types are different from each other.

For example, each aspect of a resource request can be described using an object with multiple dimensional components, or the aspect can be requested using a string. In both cases, the resource request is being described in a way that the AS needs to interpret, but with different levels of specificity and complexity for the client instance to deal with. An API designer can provide a set of common access scopes as simple strings but still allow client software developers to specify custom access when needed for more complex APIs.

Extensions to this specification can use different data types for defined fields, but each extension needs to not only declare what the data type means, but also provide justification for the data type representing the same basic kind of thing it extends. For example, an extension declaring an "array" representation for a field would need to explain how the array represents something akin to the non-array element that it is replacing.

Authors' Addresses

Justin Richer (editor)
Bespoke Engineering

Email: ietf@justin.richer.org
URI: https://bspk.io/

Aaron Parecki
Okta

Email: aaron@parecki.com
URI: https://aaronparecki.com

Fabien Imbault
acert.io

Email: fabien.imbault@acert.io
URI: https://acert.io/