Web application security: CAS and beyond

A. Maurizio Chavan*
European Southern Observatory, Karl-Schwarzschild-Str. 2, 85748 Garching, Germany

ABSTRACT

The Central Authorization Service (CAS) is used by ALMA and some of its partners (ESO, NRAO) to secure Web applications and provide Single Sign-On. CAS has been in common use throughout academia for quite some time and is well suited for securing so-called "server side" tools – that is, applications taking care of the business logic as well as generating the HTML code for the User Interface (UI).

Many Web applications are designed instead with a strong separation between a “single page” UI running in a browser and one or more back-end servers implementing the business logic; the back-ends may serve non-interactive clients, and may send requests to each other as well. Such a fragmented structure does not match CAS’ model very well and challenges system designers to come up with alternatives.

This paper describes the CAS protocol and usage, comparing it to alternative authentication and authorization models based on OAuth 2.0 that can overcome the issues CAS raises. It also tries to plot a path forward based on industry standards like OpenID Connect.

Keywords: Web Security, CAS, OAuth, JWT, JavaScript, REST, Single Page Applications, OpenID Connect

1. INTRODUCTION

The Central Authentication Service [1] (CAS) has been securing ALMA’s Web applications since the first Call for Proposals in 2011. It is been in use for ESO’s Web applications long before that, since the time it was called “Yale CAS” and was gaining popularity in the academic world.

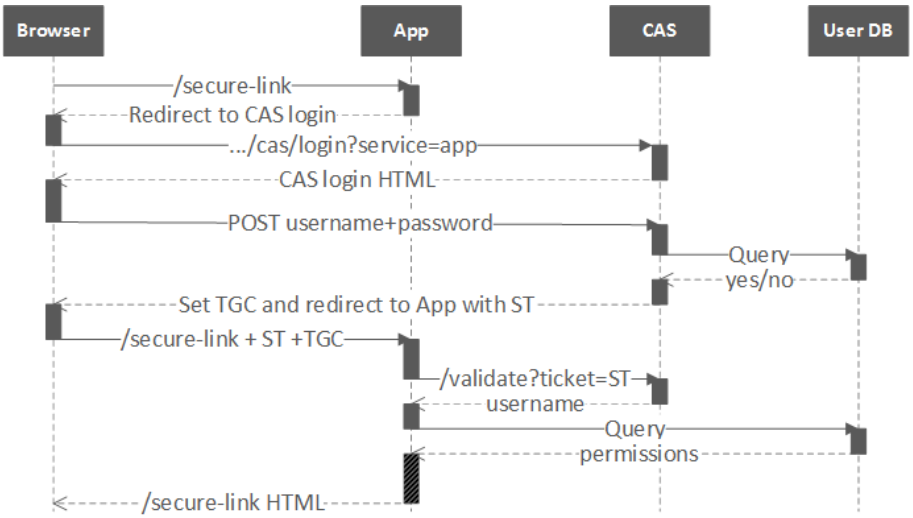

Figure 1. A simplified view of the interaction among actors in the CAS protocol, highlighting the exchange of Ticket-Granting Cookies (TGCs) and Service Tickets (STs)

* amchavan@eso.org; phone +49-89-3200-6536; www.eso.org

The CAS protocol requires the browser to interact with the application and the CAS server; application and CAS server have "back channel" interactions as well. In order to authenticate the end-user, the CAS server needs access to some user registry (for instance a dedicated database) to verify the credentials supplied on the login page.

The protocol is (deliberately) too complex to be described here; however, it is important to notice that it is browser-based, using redirects and storing cookies. Also, as the name implies, CAS is an authentication protocol: authorization is left to the applications themselves.

In the 2000's, Web applications were often written using server-side technologies like PHP, Servlets, or Zope: the application implemented both application logic and UI, and reacted to user inputs by generating a new HTML page and sending it to the user agent (browser).

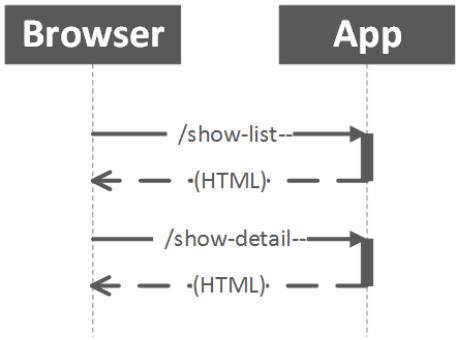

Figure 2. A simple interaction diagram showing the "page reload" model of server-side applications. Applications will often add some JavaScript to the generated pages to improve the user experience, for instance to validate user inputs.

CAS was designed with that model in mind, and does an excellent job of providing user authentication and Single Sign-On for multiple such applications; CAS clients were written for PHP, Python, Java (the original Java client became part of Spring Security) and for the Apache *httpd*.

2. SINGLE PAGE APPLICATIONS AND THEIR BACK-ENDS

Server-side applications remain very popular and are well suited for a range of domains, but require frequent page reloads that may affect the user experience. Over the last several years, the increasing availability of frameworks like Angular and React allowed small developer groups to write the kind of responsive dynamic JavaScript applications that were previously the domain of large organizations only, and the definition of the RESTful paradigm provided a simple but powerful way to exchange information with and among resource servers. The convergence of these and other factors led to an architectural pattern coupling a so-called Single Page Application (SPA) to a headless back-end implementing the business logic across one or more servers.

An SPA runs in the user browser combining JavaScript code, CSS and HTML. The "single page" qualification hints at the main departure from the UI of server-side applications: user actions cause the application to apply local modifications, without requiring a page reload. Navigation is also implemented without page reloads by selectively showing and hiding page elements. Information shown on the page is obtained by querying available back-end servers asynchronously to avoid UI "freezes" – a technique called AJAX, for Asynchronous JavaScript And XML, although the transfer format is usually JSON (JavaScript Object Notation) instead of XML. Commands are dealt with in a similar way.

A clear separation between an application's presentation layer and its business logic leads to a strong separation of concerns, allowing teams to work separately on the front- and back-end, optimizing each on their own. It also allows common functionality to be reused among applications – for instance, a reporting service – and for particularly critical components to be scaled out independently.

Back-end servers can act on behalf of actual users or batch clients, for instance a system script to perform specialized queries or routine operations. Finally, servers can obviously talk to each other as well.

Regardless of whether the back-end is “serverless” or based on “microservices”, local, distributed or Cloud-based, we have seen server-side, “monolithic” applications morph into a combination of software components running on platforms as diverse as mobile devices, embedded systems, browsers, and server farms. The Central Authentication Service was designed to secure applications, but what is an “application” in the example below?

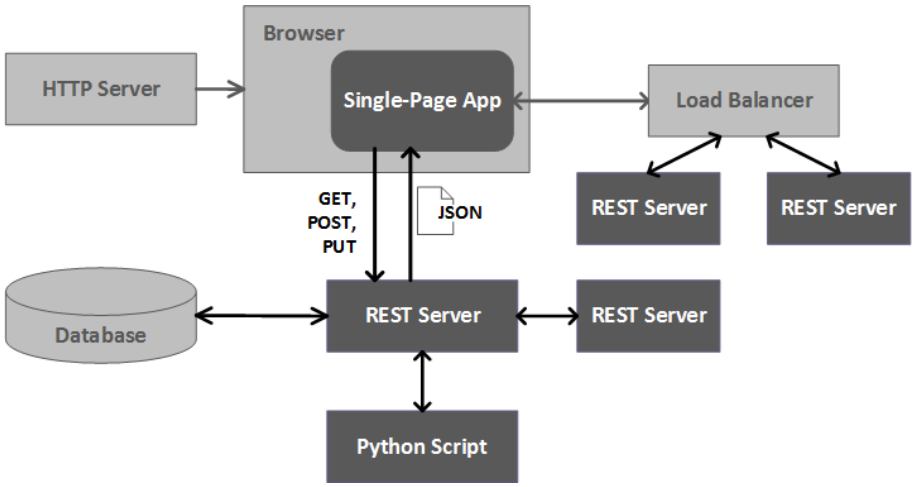

Figure 3. A hypothetical Single Page Application with a collection of RESTful back-end servers. Note the “batch” Python client and the load balancer to scale up a critical resource.

3. CAS SHORTCOMINGS AND OAUTH 2.0

It is important to repeat that the CAS protocol works very well with the Web applications it was designed for. What we see now as shortcomings of the protocol derive from a changed landscape, not from any intrinsic weaknesses. Having said that, CAS is a server-side technology that cannot be used to directly secure front-end applications. While it is possible (if painful) to authenticate the user of an SPA with CAS, the solution forces you to work around that fundamental impedance mismatch. In the same vein, a batch script or process can authenticate itself by pretending to be a user and a browser, but again, you have to work around the fact that CAS did not foresee that use case.

CAS also forces all back-ends it secures to be “stateful”, keeping track of all their logged-in user sessions. While this is not necessarily a showstopper for the kind of applications ALMA deploys to its users, it does complicate load balancing and requires extra computing resources.

Finally, native mobile applications are not supported by any of the official clients; this is not yet a direct concern for ALMA but it may become an important issue in the future.

Obviously, astronomical institutions are not the only organizations developing SPAs and having to authenticate and authorize their users. In fact, all major players in the industry have been doing that for several years using a standard called *OAuth 2.0* [2] (henceforth OAuth2) – an open protocol defined in 2012 and providing *secured delegated access* to resources. In its most general form, OAuth2 allows applications to delegate authorization to third parties (“Do you want to login with your Facebook account?”) and allows users to decide what information they are willing to share with that application (“Do you allow this application to access your email address?”).

By restricting which features are actually used, however, OAuth2 can provide functionalities very similar to CAS’. Specifically, most OAuth2 implementations can be configured to provide authentication directly, as opposed to delegating it to third-parties. Also, an internal OAuth2 implementation controls all data of its users, who implicitly allow all internal applications to access it.

4. OAUTH 2.0 ACTORS, FLOWS AND TOKENS

OAuth 2.0 is a large protocol and explaining it to any degree of completeness is outside the scope of this paper. (Many resources are available on-line for the interested reader.) In the context of a potential replacement for CAS, however, it can be presented in simpler terms introducing a few key concepts like the protocol *actors*; the message exchange patterns, called *authorization flows* or *grants*; and how *tokens* are produced and consumed. All OAuth2 tokens expire after a predefined *time to live*.

The protocol identifies four actors and assigns specific roles to each.

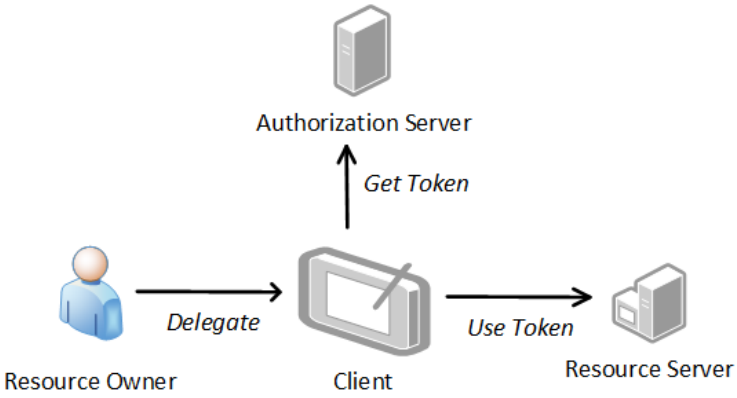

Figure 4. Actors in the OAuth 2.0 protocol

The *Resource Owner*, typically an end-user, decides if and how their private data can be accessed by a *Client* – which can be a Web or mobile application, a desktop client, a script, or any other software entity requiring secured data. The Client then requests an *access token* from the *Authorization Server*, which requires authenticating the Resource Owner. Finally, the client presents the token to a *Resource Server*, for instance a RESTful server, that validates the token and returns the data.

All clients need to be known to and registered with the authorization server. The server may be needed for validating access tokens on behalf of resource servers; alternatively, and depending on implementation choices, those servers may be configured to validate tokens directly, eliminating the need for authorization and resource servers to know about each other.

The details of message exchange patterns depend on the use cases and the nature of the client; three such flows are especially important:

- The *Authorization Code* flow requires the client to be registered with secret credentials and being able to store them securely, thus limiting its applicability of this flow to server-side Web applications and native mobile apps. It is however the most complete flow, allowing for *access* and *refresh* tokens to be used: a client in possession of a valid refresh token can use it to transparently obtain a new access token when the latter expires, thus ensuring a better user experience.
- Browser-based clients like SPAs are intrinsically less secure. The *Implicit* flow does not require secret credentials and can be used to authorize those clients. The flow does not provide refresh tokens, and applications must re-authorize when access tokens expire.
- The *Client Credentials* flow applies to non-interactive clients, for instance system scripts doing routine jobs like reporting or database cleanup. Such clients are implicitly resource owners as well (they can typically access data of any user) and are considered secure, so they can obtain an access token by presenting their secret credentials to the authorization server.

4.1 Token usage

As we have seen, access tokens are requested by a client, produced by the authorization server and consumed by resource servers; the OAuth2 specification does not stipulate any particular type of tokens, but best practices indicate that a token should be a cryptographically secure string. Depending on the flow, tokens will be returned to the client in the response body or as a URL parameter.

The OAuth2 protocol foresees that access tokens are sent by the client to the resource servers as "Bearer" tokens in the Authorization header. For instance, a *bash* client could query an ALMA resource server as follows:

```
curl -H "Authorization: Bearer AT-1-wy3AIZ" alma.cl/entities/projects?cycle=2018.1
```

Finally, a resource server requesting secured information from another resource server can pass the access token along with the request itself, achieving the goal of machine-to-machine authorization without involvement of the client.

4.2 JSON Web Tokens

The example above shows the access token passed "by reference" as an *opaque* string (that is, meaningless in itself), which needs to be validated by the authorization server. *JSON Web Tokens* (JWTs) [3] offer an interesting, "by value" alternative to that. A JWT is a means of representing *claims* to be transferred between two parties: it is digitally signed and finally encoded as an ASCII string, which can be used as an access token. Without going into the details of how a JWT is constructed, it is interesting to note that its payload is a JSON object whose fields are the claims to be exchanged, including for instance a user's ID and a set of permissions/roles. The payload representing an author and reviewer for a CMS could for instance be:

```
{"id": "mchavan", "roles": "author reviewer", "exp": 1311281970}
```

Such a token can be validated directly by the resource server, provided the latter shares with the authorization server the secret key used to digitally sign the token: the signature ensures that it comes from a trusted source and it is not been tampered with, while the *exp* claim tells the consumer whether the token is still valid or has expired. In order to minimize the risks connected with distributing sensitive information, an asymmetric encryption system is usually employed to share the secret, whereby resource servers only need to know the authorization server's public key.

In the JWT-as-access-token scenario, authorization and resource servers need to know nothing about each other beyond the shared secret key – no URLs or IDs or other locators, as JWTs are self-contained. And since access tokens with user information are submitted with every HTTP request, resource servers do not need to keep track of previous interactions and can be completely stateless.

4.3 Authentication and Single Sign-On

As previously indicated, while CAS is an authentication protocol and OAuth2 is specifically about authorization, many OAuth2 implementations in fact offer authentication services as well. With OAuth2 the line between authentication and authorization is sometimes blurred, which prompted the specification of a standard extending OAuth 2.0 to add an identity layer: *OpenID Connect* [4], known as OIDC and not to be confused with *OpenID* [5]. OIDC is a tighter spec than OAuth2: it mandates the use of JWTs as access tokens, specifies a set of standard claims, introduces the concept of ID tokens to communicate extended user information – e.g. full name and email address – and generally makes interoperability easier.

Whether based on OIDC or implementation-specific, OAuth2 authentication usually allows for Single Sign-On (SSO) as well. SSO describes the process of authenticating once for a set of related applications: it provides an improved user experience for browser users and is generally implemented by way of long-lived cookies, like CAS. In that scenario, the authorization server returns a session cookie after the first successful authentication, and subsequent authentication attempts will be silently granted if a valid session cookie comes along with the request.

5. OAUTH 2.0 AS A REPLACEMENT FOR CAS

In sec. 3 we listed some important CAS issues resulting from the popularity of new Web and mobile technologies as well as new multi-server architectures. OAuth2 bypasses those concerns as it allows authorizing different client typologies, from server-side applications to Web clients to native mobile apps to system scripts, and may be used with self-contained JWT access-tokens to implement stateless back-end configurations that can easily scale out when needed. Bearer tokens can be exchanged among back-ends to allow inter-server secured access; browser users can identify with their credentials; and Single Sign-On sessions can be established. OAuth2, with or without an OIDC implementation, represents a credible feature-by-feature replacement for CAS, without any of CAS' drawbacks related to "modern" Web applications and architectures.

For ALMA, OAuth2 would offer the added benefit of allowing distributed authentication, a long-standing requirement we were never able to implement with CAS in an efficient and simple way. ALMA's Central Authentication Service is in fact centrally located at the Santiago (Chile) Central Office, but some secure ALMA applications are deployed at its Regional Centers (ARCs) in East Asia, North America and Europe. If for any reason CAS is not accessible those Web apps cannot function, and users have been expressing the wish to use local authentication when needed or possible.

As we wrote before, authentication and resource servers do not need to know anything about each other if they share a secret key (or a pair of private/public keys). If we extend that to sharing the (private) key among authorization servers deployed at different locations, we can let a client seek authorization with any one of them and use the access token for resource servers anywhere – those servers have no way to tell where the token comes from, nor do they need to.

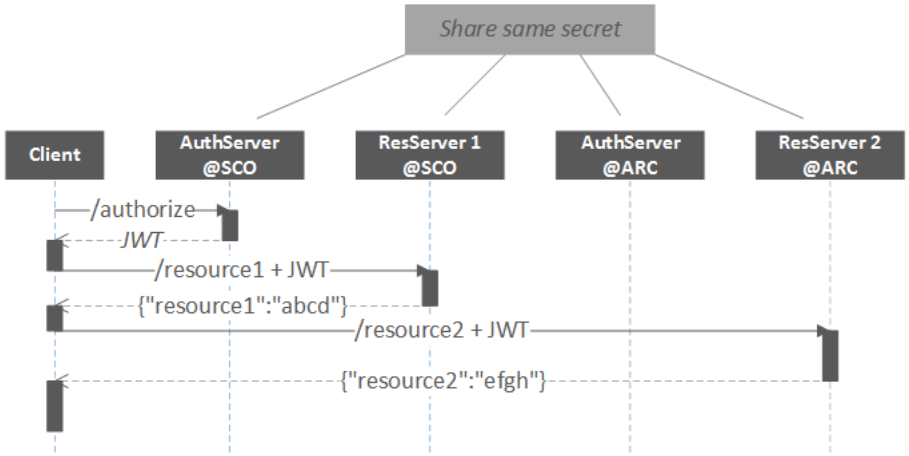

Figure 5. Example interaction among OAuth2 actors deployed across different locations:
since all servers share the same secret, JWT access tokens can be validated by any resource server

5.1 Identity federation

ALMA's user database includes accounts from many ESO and NRAO astronomers that are already registered with their own institutions. (At the time of this writing NAOJ do not have their own user registry.) That duplication has historical reasons and some concrete advantages, but forces those astronomers to manage yet another Web account. A key feature of OAuth2 is of course authorization delegation, and one could imagine buttons on a future ALMA login page offering to log in with the user's home institution account. In fact, the idea of creating a *Digital Subject* (for instance, an ALMA account) as the combination of pre-existing *Digital Identities* (NRAO or ESO accounts) is not new and is at the core of the concept of *Federated Identities* [6].

Of course, if ALMA's partner institutions were to migrate to OAuth2 as well the technical solution would be a lot simpler. In particular, the identity tokens provided by OIDC can represent the exchange medium among collaborating identity providers.

5.2 Migrate from CAS?

Even if one agrees that OAuth is a "superior" technology to CAS – at least in the specific context we have described – a legitimate question remains to be answered: why should an organization migrate from one to the other? The answer depends on the specifics of the organization, their staff and users and their current investment in CAS, but it also depends in a major way on how the organization sees its Web development efforts evolving.

ALMA finds itself in a transition phase. The bulk of its Web applications was developed using server-side technologies and use CAS, but more and more of the new development takes place as SPAs and RESTful servers that cannot use CAS and are secured with HTTP Basic Authentication [7]. As a result, users need to authenticate both via the CAS login page and the browser's Basic Authentication panel; availability of CAS and SSO depends very much on the age of the application. Although SSO has become less of a concern since browsers began offering ways to store and reuse login credentials, and sign-on is at most a mouse click away, the situation at ALMA is less than ideal and can only grow worse with time, as new non-CAS applications are deployed in production and authentication grows more and more fragmented. (SSO is not possible with Basic Authentication.) Introducing OAuth2 to replace CAS would offer a path to unify authorization and authentication once again, while offering more flexible deployment options and a path for future extensions.

Every migration exercise incurs costs and risks, which need to be weighed against the promised benefits. The following section tries to identify some of those issues.

5.3 Migrating from CAS

Replacing CAS with an alternative system like OAuth2 means setting up a different authentication/authorization server and modifying existing applications to follow the new protocol – which in turn depends on the applications' type and architecture. We'll concentrate on Java+Spring Web applications because that is what ALMA mostly develops, but other application types will be given some consideration as well.

Authentication/authorization server: there exists a single implementation of CAS, but one can choose from many OAuth2 implementations: Cloud-based or local, open-source or proprietary, with or without OIDC, and offering a wide array of additional services (like dynamic client registration) and add-ons. It is interesting to note that while many vendors offer different flavors of user management, few allow to integrate existing user databases, and the final choice of one offering over another would need to take that aspect into consideration, at least for ALMA.

Single Page Applications: when we started developing Single Page Applications with Spring Boot back-end(s) and were confronted with the issues of adapting CAS to this new model, we resorted to securing the back-ends with Basic Authentication, which requires no effort on the front-end – the browser takes care of that. (Those applications could not share Single Sign-On sessions with other ALMA apps, which gave one more reason to begin looking for a CAS replacement.) Migrating those SPAs is relatively straightforward. The front-end needs to implement the Implicit flow and there are several open-source libraries to help with that: however, one can also implement it from scratch with relative ease. (In general, OAuth2 flows are much simpler than the CAS protocol, for which it is recommended to exclusively use the official implementations: "[developing a CAS client] should be avoided as much as possible." [8])

Resource Servers: changing the Spring Security configuration of the Spring Boot-based RESTful servers from HTTP Basic to OAuth2 is also uncomplicated and well documented, by Spring itself and in dozens of blog posts. Spring Boot provides full support for OAuth2 and all that is required are a few carefully placed annotations like `@EnableResourceServer`.

"Legacy" server-side Java applications: development of our main Web applications began years ago, when Spring Boot was not available and server-side was the architecture of choice for most projects. Migrating those applications is more laborious, principally because the related documentation is much less abundant and little or no pre-packaged solutions are available. As proof of concept we converted a rather old, Spring-based, small application to use the Authorization Code flow in a couple of days, but had to write special software (servlet filters) to replace the CAS client library. Most of that time was spent looking for working examples we could adapt to our case and upgrading the application to use recent versions of Spring Security. We probably could reuse most of that work when migrating the bulk of our Web applications.

Scripts: just like our SPAs, ALMA's system scripts currently use HTTP Basic Authentication when querying our RESTful back-ends. Migrating those scripts requires acquiring an access token on startup, and implementing the Client Credentials flow is the matter of a single HTTP request. Once a token is secured, however, every request to the back-ends must include it as Bearer in the Authorization header. Depending on the size and structure of the script, that will require a certain amount of manual conversion of the code issuing the requests.

Non-Java, server-side Web applications: the ALMA Science Portal is implemented on top of Plone [9], a Python-based CMS and application development environment. There is no official Python CAS client and the Portal is integrated with CAS by way of a community-provided client; the user experience is less than optimal. Several Python libraries can be used to integrate a Web application with OAuth2, but as is the case with JavaScript, implementing a grant type from scratch is also doable. In the case of the Portal, which acts as an entry point for public information and user-facing applications and can be trusted with storing secret client credentials, the Authorization Code flow would be a good choice.

Desktop Java applications: none of our desktop Java applications authenticates with CAS, but it would be possible to integrate them with an OAuth2 authentication/authorization system. Depending on the application type, the *Resource Owner Password Credentials* grant would be a good candidate for implementation, as it does not require a Web-based form for collecting user credentials. A system-type application could use the Client Credentials flow instead.

5.4 Migration strategies: progressive conversion and hybrid authentication

While migrating from CAS to OAuth2 would be a non-trivial exercise, it would not present special technical hurdles. However, the actual process of migrating an entire working system without disrupting Observatory operations would be rather delicate.

A progressive approach would probably be required, with CAS and OAuth2 operating in parallel for a limited period of time while applications are ported over, until CAS can be decommissioned. Obviously, there would be no Single Sign-On across the two systems during the transition period.

Another interesting possibility is offered by CAS itself, which can be configured to support the OAuth2 protocol and OIDC in addition to, and in parallel with, CAS' own. As proof of concept we deployed a server-side, JSP-based application configured for CAS and an SPA configured for the Implicit flow, both authenticating against the same server. One could sign on with either application and be implicitly authenticated on the other, achieving inter-protocol SSO (with some glitches).

On the down side, CAS' implementation of OAuth (as of version 5.2) is incomplete, as it cannot be configured to generate JWTs as access tokens. (It may be in fact possible, but it is unclear how to from the available documentation. The glitches with cross-protocol SSO may also be due to documentation issues.) The OIDC implementation returns opaque access tokens as well, although the standard mandates well-defined JWTs.

A migration strategy based on deploying a hybrid CAS/OAuth2 authentication server could provide organization-wide SSO during the transition period, but that advantage needs to be weighed against the disadvantage of a sub-standard implementation of the OAuth protocol.

6. CONCLUSIONS

New Web technologies and system deployment models represent a challenge for astronomical institutions like ALMA, which have been using the Central Authentication Service since they began deploying Web applications. Major industry players have converged on OAuth 2.0 as a secure delegation protocol covering a wide array of Web, mobile and command-line application types. By using a subset of that protocol and taking advantage of the many authentication extensions available as open-source or paid tools, OAuth2-based solutions can replace CAS authentication, overcoming its limitations and opening the way to further extensions, like distributed authentication and federated identities.

While migration from CAS to an alternative solution would not be a trivial task, progressive or hybrid strategies can be used to limit the operational risks associated with a "big-bang" approach and spread the work over an acceptable time range. Organizations like ALMA need to look at the technical and operational challenges vs. the risks and benefits

associated with such a migration, and decide if and how to update their authentication/authorization infrastructure. In any case, OAuth2 represents a technically sound and accessible alternative to CAS.

ACKNOWLEDGEMENTS

The author would like to thank his colleagues at ESO and ALMA for the suggestions and criticism received during the weeks and months that led to writing this paper. Without their ideas, insight and participation this work would not have been possible. The authors consider himself fortunate to have so many supportive colleagues; unfortunately, they are too many to be listed by name.

REFERENCES

[1] CAS, https://www.apereo.org/projects/cas
[2] The OAuth 2.0 Authorization Framework, https://tools.ietf.org/html/rfc6749 (2012)
[3] JSON Web Token (JWT), https://tools.ietf.org/html/rfc7519 (2015)
[4] OpenID Connect Core 1.0, http://openid.net/specs/openid-connect-core-1_0.html (2014)
[5] OpenID Authentication 2.0, https://openid.net/specs/openid-authentication-2_0.html (2007)
[6] Federated identity, https://en.wikipedia.org/wiki/Federated_identity
[7] HTTP authentication, https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication
[8] CAS Clients, https://apereo.github.io/cas/4.2.x/integration/CAS-Clients.html
[9] Plone, https://plone.com/