ESUP-Portail: open source Single Sign-On with CAS (Central Authentication Service)

Pascal Aubry*, Vincent Mathieu†, Julien Marchal†

* IFSIC, University of Rennes 1
pascal.aubry@univ-rennes1.fr

† University of Nancy 2
vincent.mathieu@univ-nancy2.fr, julien.marchal@univ-nancy2.fr

Abstract

The universality of the HTTP protocol seduced developers for quite long; most applications are web-based today.

LDAP directories saved our users' brains by making them memorize only one password, but their fingers are still very much in demand by all the authentications they need to type, in practice each time they access an application.

Many solutions for Single Sign-On are already available. We describe here a free, simple, complete and sure solution: CAS (Central Authentication Service), developed by Yale University. CAS has been chosen by the French ESUP-Portail consortium, which provides a complete and opened solution to Universities and University-level colleges to offer an integrated access to their services and information for their students and staff.

Keywords: Single Sign-On, open-source, authentication.

1 Why do we need Single Sign-On?

Web-based applications (mailers, forums, agendas, other specific applications) have largely spread over our networks during the last years. These applications often need authentication.

The use of LDAP directories allowed a single account for our users, which is obviously a real improvement. Some issues however remain:

- **Multiple authentications**: giving netId/password to each application is still needed;
- **Security**: as user accounts are unique, password stealing is really critical; the security of the authentication process is essential. Moreover, user credentials should not be given to applications any more.
- **Several authentication mechanisms**: some users own personal X509 certificates [1], which can be used for authentication. Moreover, even if LDAP is a widely used standard today, it could at least be replaced by other user databases; Anyway, abstracting the authentication mechanism(s) is interesting, for instance to be able to use mixed authentication.
- **Cooperation**: transparently accessing resources of one establishment when only authenticated against another one is a wish for close institutions, especially in our educational community.
- **Authorization**: applications often need to know users' profiles to allow (or deny) them to perform specific actions.

The principle of all SSO solutions is to remove authentication from applicative code. The goal is then to offer a globally secured software environment:

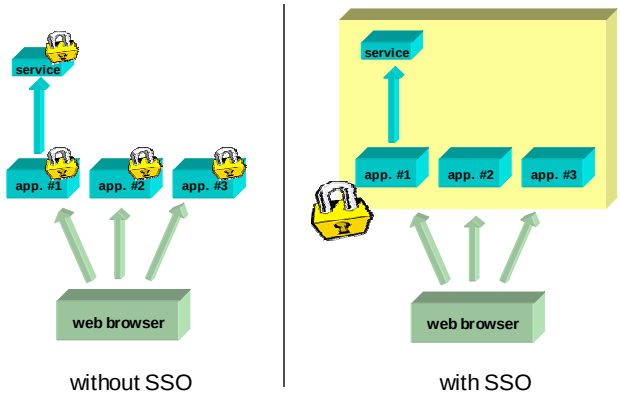

Figure 1: The principle of Single Sign-On

SSO mechanisms [2] try to answer these questions using similar techniques:

- **Authentication is centralized** to a unique server, the only machine receiving users' credentials, through an encrypted tunnel;
- **HTTP redirections are used**, from applications to the authentication server for unauthenticated users, and back to applications when authenticated;
- **Information is passed** by the authentication server to applications during the redirections, thanks to cookies [3] and/or CGI parameters.

Among the commercial solutions offered to system administrators and developers, two leaders stand out: Sun One Identity Server [4] and Microsoft Passport [5].

After having tested several free implementations (the ESUP-Portail project is based on open-source software only), the ESUP-Portail SSO group chose CAS (Central Authentication

Service [6], developed by Yale University) for its Single Sign-On mechanism.

2 The reasons why we chose CAS

CAS is made up of java servlets, and, run over any (JSP spec 1.2 compliant) servlet engine, offers a web-based authentication service. Its strong points are security, proxying features, flexibility, reliability, and its numerous client libraries.

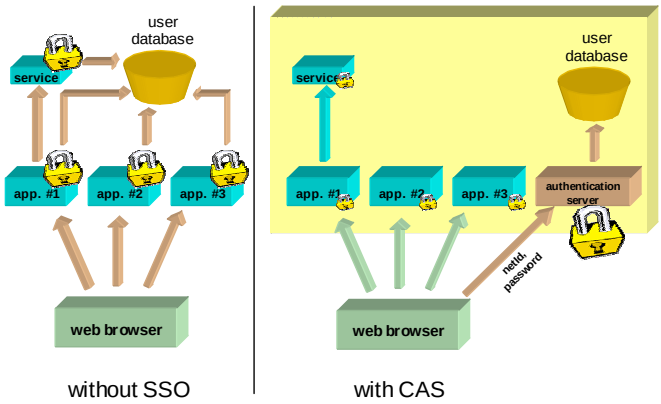

Figure 2: A software environment secured by CAS

2.1 Security

Security is insured the following ways:

- Passwords only pass from browsers to the authentication server, always through an encrypted tunnel;
- Re-authentications are transparent to users, providing that they accept a single cookie, called the 'Ticket Granting Cookie' (TGC). This cookie is opaque (no personal information), protected (HTTPS) and private (only presented to the authentication server);
- Applications know users' identities thanks to opaque one-time 'Service Tickets' (ST). Those tickets are emitted by the authentication server, transmitted to applications by the browsers, and finally validated by the authentication server (returning the corresponding identity). This way, applications never see any password (as it is the case for almost all serious SSO mechanisms).

2.2 Authentication proxying

Classical SSO mechanisms demand a communication between the browser and the application, which forbids multi-tier installations, where an application must request a back-end service needing authentication (for instance a portal requesting a web service).

CAS v2.0 solves this issue by proposing an elegant way to propagate authentication without propagating passwords; dedicated tickets (PGT: Proxy Granting Ticket and PT: Proxy Ticket) allow third-party applications to get sure of users' identity. This feature is obviously the strongest point of CAS.

2.3 Flexibility

The package proposed by CAS developers offers a complete implementation of the authentication protocol, but the authentication itself (against a user database) is left to the administrator. We wrote a generic handler which provides several connectors (LDAP, X509 certificates, NIS domains, databases) which can be used alone, or together to get mixed authentication. This generic handler can also been extended to give system administrators other authentication methods, such as Kerberos or Active Directory.

2.4 Client libraries

The code handling the basic protocol (apart from proxying) is very simple to write on the client-side (applications). **Client libraries** were provided for Perl, Java, ASP and PL/SQL. We added a strong (proxy-able) PHP library. These libraries give a very impressive flexibility to CAS-ify existing applications by simply adding a few lines of code.

An **Apache** module (mod_cas) lets web servers authenticate users for static resources, as client libraries can not be used in this case.

A PAM (**Pluggable Authentication Module** [7]) module (pam_cas) allows the integration of non web-based applications at a very low level.

2.5 Moreover…

CAS is used by many American Universities, with LDAP or Kerberos-based authentication. This makes us confident in its **permanence**.

At least, CAS can be directly plugged into **uPortal** [8], chosen by the ESUP-Portail consortium, on the way to become a standard for open source portals.

This article shows how Single Sign-On is achieved with CAS, and focuses on a precise technical issue: CAS-ifying a webmail (Horde IMP) and an IMAP server (Cyrus IMAP).

3 How CAS works

3.1 Architecture

3.1.1 The CAS server

Authentication is centralized on a unique machine, called the CAS server. This machine is the only actor knowing users' passwords. It has a double role:

- Authenticating users;
- Transmit and certify the identities of authenticated users (to CAS clients).

3.1.2 Web browsers

Web browsers should meet the following requirements to take advantage of all CAS comfortable features:

- Own an encryption engine to be able to use HTTPS;
- Perform HTTP redirections (access a URL given by a Location header when receiving 30x responses) and understand basic Javascript;
- Store cookies, as defined in [3]. In particular, for security purposes, private cookies should be transmitted only to the machines that emitted them.

These requirements are met by all classical web browsers, i.e. Microsoft Internet Explorer (since 5.0), Netscape Navigator (since 4.7) and Mozilla.

3.1.3 CAS clients

A web application equipped with a CAS client library, or a web server using mod_cas, is called a CAS client. It delivers resources only to clients previously authenticated by the CAS server.

CAS clients are:

- Libraries, corresponding to the most widely used web programming languages (Perl, Java, JSP, PHP, ASP);
- An Apache module, used in particular to protect static documents;
- A PAM module, used to perform system level authentication.

3.2 Basic functioning

3.2.1 Authenticating a user

A non previously-authenticated user (or a user of which authentication expired) accessing the CAS server is presented an authentication form, in which (s)he is invited to enter a netId and a password:

Figure 3: First access of a browser to the CAS server

If netId and password are correct, the server sends a cookie called TGC (Ticket Granting Cookie) to the browser:

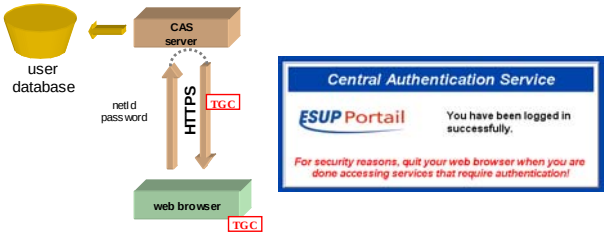

Figure 4: Authentication of a browser against the CAS server

The TGC is the user's passport against the CAS server. Its lifetime (validity) is limited (typically a few hours). It is the way for web browsers to get tickets (meant for CAS clients) from the CAS server, without needing to re-authenticate. It is a private cookie (never transmitted to web servers but the CAS server), and protected (requests to the CAS server are secured). As all the tickets played with CAS, it is opaque (i.e. contains no information on the user): it is just a session identifier between the web browser and the CAS server.

3.2.2 Accessing a protected web resource when authenticated

When accessing a resource protected by a CAS client, the web browser is redirected to the CAS server. The browser, previously authenticated, provides the CAS server its TGC:

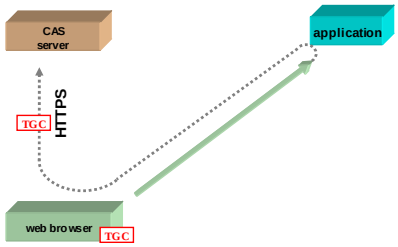

Figure 5: Redirection of an unknown browser to the CAS server

On presentation of the TGC, the CAS server delivers a Service Ticket (ST). It is an opaque ticket (no user information), and is usable only by the service that required it. At the same time, the CAS server redirects the browser to the calling service (the Service Ticket is a CGI parameter):

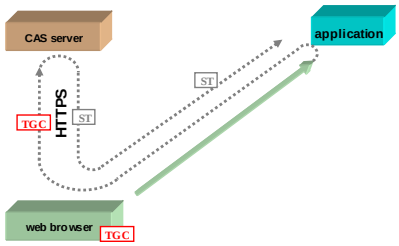

Figure 6: Redirection of the browser to the calling service after authentication

The ST is then validated by the CAS client against the CAS server (thanks to an HTTP request) and the wanted resource can be delivered to the browser:

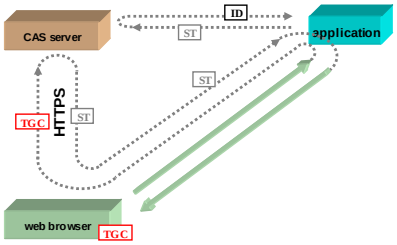

Figure 7: Validation of a Service Ticket

Let us remark that all the redirections above are transparent for the user: he accesses the resource without authenticating, and without interacting at all.

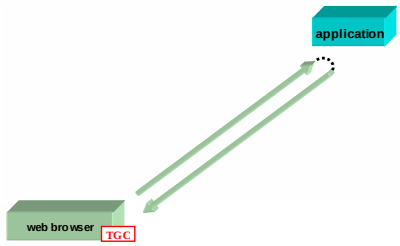

Figure 8: User view of the CAS redirections

The Service Ticket (ST) is the browser's passport against a CAS client. It is a One-Time Ticket (can not be presented twice to the CAS server), valid only for the CAS client it was delivered for, and only for a very short period of time (typically a few seconds).

3.2.3 Accessing a protected resource when not authenticated

If the browser was not previously authenticated, it is redirected to the CAS server, which returns an authentication form.

When correctly authenticating by submitting the form, the CAS server:

- Sends the browser a TGC, that will exempt it from re-authenticating later;
- Redirects the browser to the calling service (the CAS client), with a Service Ticket.

As you can see, there is no need to be previously authenticated to access a protected resource: authentication is automatically demanded the first time a user requests a protected resource.

3.3 Multi-tier configuration

3.3.1 CAS proxies

CAS multi-tier feature brings the possibility for a CAS client to access a back-end service under the primarily authenticated user's identity. Such a CAS client, able to proxy credentials is therefore called a CAS proxy. Most used CAS proxies are:

- Web **portals**, which need to access external applications (web services [9] for instance) under users' identities;
- **Webmail** applications, which need to connect to an IMAP server to retrieve email under users' identities.

In a multi-tier CAS installation, CAS clients do not have access to the browser's cookie cache any more, and redirections can not be used.

3.3.2 2-tier functioning

A CAS proxy, when validating a Service Ticket to authenticate a user, also enquires a PGT (Proxy Granting Ticket) from the CAS server:

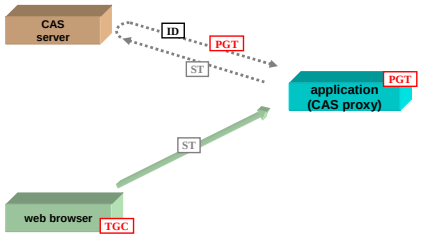

Figure 9: PGT retrieval by a CAS proxy

A PGT is a CAS proxy's passport, for a user, against the CAS server. It is the way for CAS proxies to get tickets (meant for CAS back-end services) from the CAS server, without needing to validate a ST. It is an opaque and re-playable ticket, delivered by the CAS server through a secured request, to insure its integrity and confidentiality. PGTs' lifetime is limited (a few hours, as well as TGCs).

PGTs are for applications the equivalent of TGCs for web browsers. A PGT allows applications (CAS proxies) to authenticate a user against the CAS server, and get Proxy Tickets (PTs are for CAS proxies the equivalent of STs for web browsers). Proxy Tickets are, as well as Service Tickets, validated by the CAS server before giving access to protected resources:

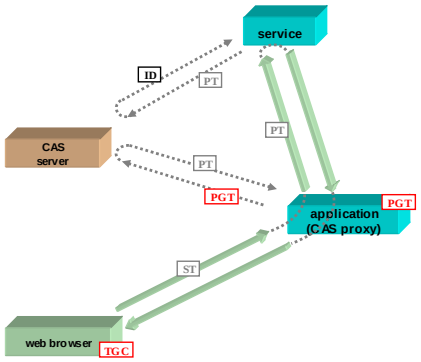

Figure 10: Validation of a Proxy Ticket by a back-end service

3.3.3 N-tier functioning

It is easy to see the back-end service accessed by the CAS proxy in 2-tier configuration can be a CAS proxy itself. CAS proxies can be chained:

Figure 11: A chain of CAS proxies

CAS is, at this time, the only free SSO mechanism allowing such n-tier installations without propagating any user password.

4 CAS user authentication

The original CAS distribution does not include user authentication. Authentication classes have to be written by administrators, and fit to their exact need (some example classes are provided).

4.1 The GenericHandler class

Developed by the ESUP-Portail project [10], the GenericHandler class [11] provides the implementation of many authentication methods: LDAP directories, databases, NIS, files, NT domains, etc. Furthermore, this class can be easily extended to fit other needs (Novell, Kerberos, Active Directory, etc.).

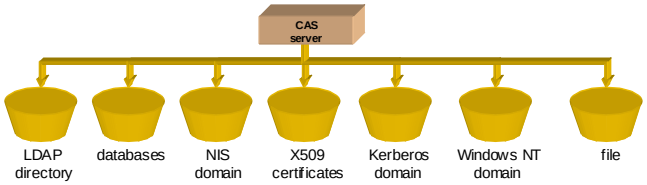

Figure 12: User authentication with ESUP-Portail GenericHandler

The configuration is done in an XML format: one or more authentication methods are specified. They will be sequentially tested until one succeeds.

Because LDAP became a standard for storing and authenticating users, we show, as an example, how Generic Handler can be used with an LDAP directory.

4.2 Authentication against an LDAP directory

Two different access modes are proposed, depending on the internal structure (DIT) of the LDAP directory.

4.2.1 Direct access mode (ldap_fastbind)

ldap_fastbind mode can be used against LDAP directories of which users' DN (Distinguished Name) can be directly deduced from their netId (in practice, directories where users are stored at the same hierarchical level, in the same OU for instance).

In this case, CAS tries to connect the directory using the DN and password provided by the user. Classically, the user is authenticated if the connection succeeds.

One may use:

```
<authentication>
  <ldap version="3" timeout="5">
    <ldap_fastbind filter="uid=%u,dc=univ-rennes1,dc=fr" />
    <ldap_server host="ldap.ifsic.univ-rennes1.fr"
                 port="389"
                 secured="no" />
  </ldap>
</authentication>
```

4.2.2 Search mode (ldap_bind)

When the DN can not be deduced from the uid, administrators must use the ldap_bind mode, with which the user's DN is searched before attempting a connection. For instance:

```
<authentication>
  <ldap version="3" timeout="5">
    <ldap_bind search_base="dc=univ-rennes1,dc=fr"
               scope="sub" filter="uid=%u"
               bind_dn="admin" bind_pw="secret" />
    <ldap_server host="ldap.ifsic.univ-rennes1.fr"
                 port="389" secured="no" />
  </ldap>
< /authentication>
```

4.2.3 LDAP Redundancy

Generic Handler can use redundancy to be more fault-tolerant: it is possible to specify a list of LDAP servers, which are considered as replicas.

5 CAS-ifying a web application

CAS-ifying a web application is very easy, thanks to CAS client libraries.

Three kind of CAS applications exist:

- **CAS "simple" clients**: they only need to authenticate users.
- **CAS proxies**: they need to authenticate users, but also use tier services. They need to retrieve PGTs from the CAS server, and later PTs they will transmit to back-end services to authenticate the users they act for.
- **CAS back-end services**: they need to validate PTs given by CAS proxies and get users' identities.

5.1 "simple" CAS clients

The principle is to use a function (or method) which will run the authentication mechanism and return the user's netId. This function must perform the following tasks:

- If the user is not already authenticated and no ST is provided, redirect the web browser to the CAS server (providing its own URL for coming back later);
- If the user is not already authenticated and a ST is provided, validate the ST by using an HTTPS request to the CAS server. The CAS server should then return the corresponding user's netId.

To illustrate the simplicity of the CAS-ification of such a "simple" CAS client, we show below how a CAS client can be written in PHP. Of course, in a real application, a client library, like phpCAS [12] in our case, should be used instead.

5.1.1 Writing a PHP CAS client

If this script (`script.php`) is called without any parameter, it redirects the web browser to the CAS server, giving its own URL as a CGI parameter:

```
https://cas.univ.fr/login?service=http://test.univ.fr/script.php
```

The user authenticates against the CAS server, which redirects the browser to the calling service, giving a ST as a CGI parameter. The coming-back URL would be something like:

```
http://test.univ.fr/script.php?ticket=ST-2-
uw2KEWinSFeZ9fotZIio
```

Our script will then try to validate the Service Ticket against the CAS server, by accessing the following URL:

```
http(s)://auth.univ.fr/serviceValidate?service=http://test.
univ.fr/script.php&ticket=ST-2-uw2KEWinSFeZ9fotZIio
```

The CAS server validates the ticket and returns the user's netId, in an XML response:

```
<cas:serviceResponse
xmlns:cas='http://www.yale.edu/tp/cas'>
  <cas:authenticationSuccess>
     <cas:user>paubry</cas:user>
  </cas:authenticationSuccess>
</cas:serviceResponse>
```

A possible implementation of this script is:

```
<?php /* PHP simple Cas client */
// localization of the CAS server
define('CAS_BASE','https://auth.univ.fr');

// own URL
$service='http://'.$_SERVER['SERVER_NAME']
                  .$_SERVER['REQUEST_URI'];

/** Authenticate against a CAS server
 * @return the user's netId, or FALSE on failure
 */
function authenticate() {
  global $service;

  // retrieve the ticket
  if (!isset($_GET['ticket'])) {
    header('Location:
'.CAS_BASE.'/login?service='.$service));
    exit();


  // try to validate the ST against the CAS server
  $fpage = fopen (CAS_BASE . '/serviceValidate?service='
                . preg_replace('/&/','%26',$service)
                . '&ticket=' . $ticket, 'r');
  if ($fpage) {
    while (!feof ($fpage)) { $page .= fgets ($fpage, 1024);
  }

  // analyze the CAS server's response
  if (preg_match('|<cas:authenticationSuccess>|mis',
                 $page)) {
    if(preg_match('|<cas:user>(.*)</cas:user>|',
                  $page,$match)){
      return($match[1]);
    }
  }

  // validation failed
  return FALSE;
}

if (($login = authenticate()) === FALSE ) {
  echo 'failure (<a href="'.$service.'">Retry</a>).';
  exit() ;
}

echo 'welcome user '.$login'!<br>'
echo '(<a href="'.CAS_BASE.'/logout"><b>logout</b></a>)';
?>
```

5.1.2 Using the phpCAS client library

The phpCAS library [12] was developed by the ESUP-Portail project. Here is the way it can be used:

```
<?php  /* a simple CAS client using phpCAS */
include_once('CAS.php');
phpCAS::client(CAS_VERSION_2_0,'cas.univ.fr',443,'');
phpCAS::authenticateIfNeeded();
?>

<html>
  <body>
    <h1>Authentication succeeded!</h1>
    <p>User is <?php echo phpCAS::getUser(); ?></b>.</p>
 </body>
</html>
```

5.2 CAS proxies

The procedure exactly begins as for "simple" CAS clients: retrieve a Service Ticket.

Next, when validating the ST, an additional parameter is given to the CAS server: a callback URL. In response, the CAS server returns:

- The user's netId (as for an ordinary CAS client);
- A PGT, using the callback URL.

As seen in 3.3.2 ("2-tier functioning"), the PGT will be used later to authenticate a user against the CAS server and get Proxy Tickets needed to access back-end services.

Java and PHP libraries mask the complexity of all this when developing a CAS proxy. Here is, for instance, the way a CAS proxy can be implemented thanks to the phpCAS library:

```
<?php /* a CAS proxy using phpCAS */
  include_once('CAS.php');
  phpCAS::proxy(CAS_VERSION_2_0,'auth.univ.fr',443,'');
  phpCAS::authenticateIfNeeded();
?>

<html><body>
<p>User's netId: <?php echo phpCAS::getUser(); ?>.</p>
<?php
  flush();
  if (phpCAS::serviceWeb('http://test.univ.fr/ws.php',
                         $err_code, $output)) {
    echo $output;
  }
?>
</body></html>
```

5.3 CAS back-end services

Back-end services are as easy to CAS-ify as "simple" CAS clients because they do exactly the same job, i.e. validating a Proxy Ticket (instead of Service Ticket) against the CAS server.

The back-end service called by the CAS proxy shown before could be:

```
<?php  /* a simple CAS back-end service */
  include_once('CAS.php');
  phpCAS::client(CAS_VERSION_2_0,'cas.univ.fr',443,'');
  phpCAS::authenticateIfNeeded();

  echo '<p>User is ' . phpCAS::getUser() . '.</p>';
?>
```

5.4 Precautions to take when CAS-ifying web applications

5.4.1 Sessioning

Applications should maintain sessions for the CAS mechanism not to be fired at each request, but only once, for evident performance issues.

This remark goes for CAS clients and proxies (that should maintain a session with the browser) as well as for back-end services (that should maintain a session with the CAS proxy).

5.4.2 Asynchronism

Retrieving a PGT for a user in a CAS proxy is easy, when using CAS client libraries. Developers should however take care of possible desynchronizations between the different sessions of a multi-tier CAS installation.

Let us explain this with an example. A user connects in a web portal, which will act as a CAS proxy: the user authenticates against the CAS server, the portal retrieves a PGT for the user, and a session is set between the portal and the browser. This session is set to last a few hours.

Let us now imagine that the PGT becomes invalid (expiration or user logout from another window of the browser). In this particular configuration, it is impossible for the portal to get new PTs and thus access back-end services.

This situation should be handled by CAS proxies, for instance by forcing the disconnection of the user.

5.5 CAS authentication for static web pages

The CAS mechanism can be used to protect static resources (typically HTML web pages), thanks to the mod_cas Apache module.

With simple Apache directives, the access to a site (or part of it) can require an authentication against a CAS server. For instance, the following directives will redirect users to the CAS server located at https://cas.univ.fr/cas if no valid ST is given by browsers:

```
CASServerHostname   cas.univ.fr
CASServerPort       8443
CASServerBaseUri    /cas
CASServerCACertFile /etc/x509/cert.root.pem

<Location /protected>
    AuthType CAS
    Require valid-user
</Location>
```

6 CAS-ifying a non-web application

The main goal of an SSO mechanism is of course to provide a unique authentication service for web applications, efficient and simple. CAS offers more by allowing the CAS-ification of non-web services, such as IMAP, FTP, etc.

In order to do this, these services should use PAM (Pluggable Authentication Module), as most Unix services do now.

6.1 The PAM pam_cas module

Pam_cas is included into CAS client distribution. It is powerful and however light (about 300 lines of C, half of them shared with mod_cas).

It allows a service to authenticate a user by receiving an identifier (a netId, as usually) and a ticket (instead of a password). The ticket received by the service is then validated by pam_cas against the CAS server.

Let us notice that using pam_cas can not be thought outside of a multi-tier installation: the CAS-ified service must be accessed by a CAS proxy. Indeed, it is unconceivable to ask a human being (human user of an FTP service for instance) to provide a CAS ticket.

Fortunately, PAM modular concept allows us to use pam_cas in conjunction with other PAM modules. It is possible for a service to authenticate user in a traditional way like they used to do (netId and password) or with CAS method (netId and ticket) at the same time.

The example below shows how this can be done.

6.2 Using pam_cas to CAS-ify an IMAP server

Our goal is here to CAS-ify an IMAP server, to set connections from a web portal (with Proxy Tickets), while continuing to accept connections from traditional mail clients (with passwords).

If the IMAP server is PAM-compliant (which is generally the case), the PAM configuration can look like:

```
auth sufficient /lib/security/pam_ldap.so
auth sufficient /lib/security/pam_pwdb.so shadow nullok
auth required   /lib/security/pam_cas.so \
     -simap://mail.univ.fr \
     -phttps://ent.univ.fr/uPortal/CasProxyServlet
```

In this example, authentication will be first attempted against an LDAP directory, next against the local Unix user database, and finally with pam_cas: the secret provided is validated against the CAS server (internally, only if it is ticket-shaped for evident performance issues).

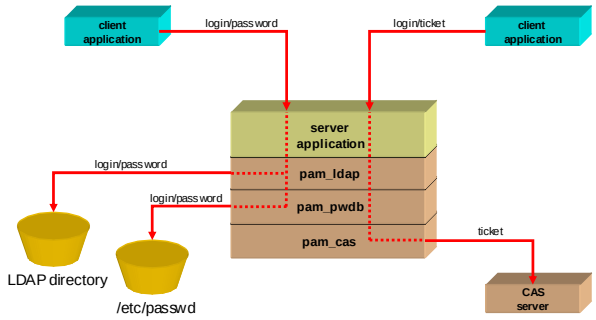

Figure 13: Using pam_cas

6.3 CAS-ifying the Cyrus-IMAP server

The IMAP protocol is very particular, and probably the most difficult to CAS-ify. IMAP clients and mainly webmails have

the odd habit to generate very numerous requests, closing and re-opening connections very often. This of course leads to numerous authentication requests against the CAS server.

When using a traditional webmail (on which users authenticate with their netId and password), the only consequence is a heavier load for the web server running the webmail. Within a CAS multi-tier installation, load increase is supported by the web server running the webmail, but also by the CAS server.

This is clearly prohibitive, for performance issues, to ask for a ticket and validate it at each request: as a consequence, a cache is needed on the IMAP server (to make the PT re-playable by the webmail).

The implementation of such a cache comes straight with Cyrus. Indeed, Cyrus IMAP server uses Cyrus-SASL for authentication; now, Cyrus-SASL can use different authentication mechanisms (PAM, LDAP, Kerberos, etc.) or call a Unix daemon, saslauthd.

This daemon, which communicates with Cyrus-SASL thanks to a Unix socket, proposes a cache mechanism. Thanks to this cache, the mail client will be able to play the same PT more than once, because saslauthd will not use PAM once the ticket is stored in its cache.

CAS-ifying Cyrus-IMAP this way made us save 95% of the authentication requests (only 5% were really played, i.e. tickets validated against the CAS server).

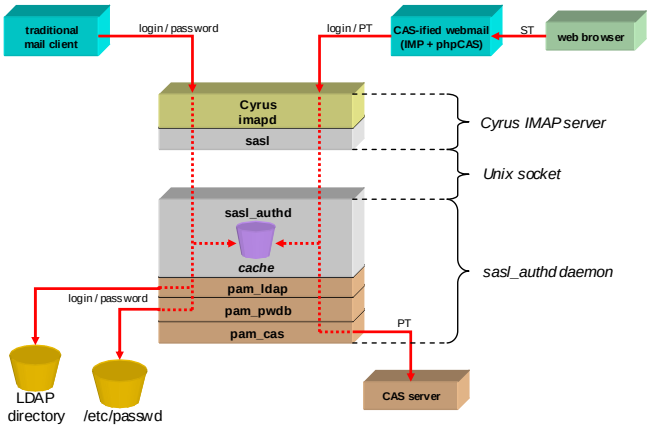

Figure 14: CAS-ification of Cyrus-IMAP

6.4 CAS-ifying Horde IMP

Our primary goal was to integrate a webmail product into ESUP-Portail software, if possible completely integrated within our SSO. We decided to do it with Horde IMP [13].

At first, IMP was adapted to become a CAS proxy. This was easily done by using the phpCAS library, as seen in 5.2 (“CAS proxies”). It was then possible to acquire a Proxy Ticket and make the IMAP server authenticate users, by validating PTs against the CAS server.

Next, the behavior of the webmail was modified to take into account the versatility of this new kind of password. Indeed, PTs are manipulated the same way passwords are, but their lifetime is limited. In other words, the webmail can use a PT several times thanks to the IMAP cache, but the PT stored in the IMAP cache can be erased (because of the garbage collector of the IMAP cache), or replaced in the cache by another PT (if another webmail instance is running for the same user), or simply replaced by the user’s password if the user concurrently uses a traditional mail client. In this case, the next connection with the PT would be refused by the IMAP server. To get round this problem, the webmail was modified to acquire a new PT from the CAS server, and try to make an IMAP connection a second time.

You probably guess now that CAS client libraries are not as simple as we said in 5.1.1 (“Writing a PHP CAS client”).

7 Restrictions and perspectives

We described in this article the strong points of CAS:

- It is an open-source and free product;
- Its security level is very satisfying;
- A CAS server is very easy to set up and configure;
- Web applications are very easy to CAS-ify.

Now we see CAS limits, as well as perspectives to get round these delicate points.

7.1 CAS brings SSO, nothing else

CAS is proposed as a Single Sign-On mechanism, and we saw that it can also run at system-level, thanks to pam_cas. On the other hand, it is strictly limited to user authentication: it does not (and probably will never) deal neither with authorizations nor with the propagation of user attributes.

Moreover, user databases are local, at the establishment-level. Multi-establishments issues are not addressed by CAS. Recent developments on Sympa [14] show an elegant way to allow authenticating users from several establishments, by relying on several CAS servers. However, the most promising way to make different establishments cooperate with CAS is certainly the Shibboleth internet2 project [15].

7.2 Performance and fault-tolerance

In a CAS installation, all the web applications depend on the CAS server. Its availability is critical.

In its current release, load balancing can not be implemented. Indeed, CAS tickets are stored by the CAS server into memory, for efficiency and simplicity. This makes impossible to share between several CAS servers.

In practice, Universities having deployed CAS never encountered performance issues, certainly because processes involved are quite light. On the other hand, the absence of fault tolerance is much more crucial, as the CAS server really becomes a pivot of the web software suite of an establishment.

It is of course possible to maintain a sleeping spare server, which can be used in case of failure, or more simply for maintenance. Switching between two Tomcat servers behind an Apache frontal is really easy, and this is solution recommended by the ESUP-Portail consortium. However, this solution is not transparent for connected users: all valid tickets (especially TGCs and PGTs) are lost.

A solution, consisting in storing granting tickets (TGCs and PGTs) into a database is conceivable. In this case, switching from one CAS server to another one would have very limited effects (only STs and PTs would be lost), while preserving simplicity and thus performance.

8 What about CAS in the future?

We are very confident in CAS. Adopted by the ESUP-Portail consortium as its SSO software, CAS will in the coming months be deployed in all the French Universities that chose ESUP-Portail software. We strongly believe that it can become a standard.

The ESUP-Portail consortium takes an active part in popularizing CAS, notably by distributing a CAS server quick-start, which allows any system administrator to setup and configure a CAS server in a few minutes.

References

[1] Autorité de certification du CRU, in french, http://pki.cru.fr

[2] Single Sign-On architectures, Jan de Clercq, RSA2003, November 2003, Amsterdam, http://www.rsaconference.com/rsa2003/europe/tracks/pdfs/implementers_w14_declercq.pdf

[3] Persistent client state (HTTP cookies). http://wp.netscape.com/newsref/std/cookie_spec.html

[4] Sun One Identity Server. http://www.sun.com

[5] Microsoft .NET Passport: One easy way to sign in online. http://www.passport.com

[6] ITS Central Authentication Service, http://www.yale.edu/tp/cas/

[7] Linux-PAM: Pluggable Authentication Modules for Linux, www.us.kernel.org/pub/linux/libs/pam/Linux-PAM-html/

[8] JASIG (Java Architectures Special Interest Group), Evolving portal implementations. http://mis105.mis.udel.edu/ja-sig/uportal/

[9] Web Services, http://www.w3.org/2002/ws/

[10] ESUP-Portail, http://www.esup-portail.org

[11] CAS GenericHandler, http://esup-casgeneric.sourceforge.net

[12] PhpCAS, http://esup-phpcas.sourceforge.net

[13] The Horde Project, http://www.horde.org

[14] Authentication and access control in Sympa mailing list server, Serge Aumont & Olivier Salaun, TERENA2004, June 2004, Rhodes, http://www.sympa.org

[15] The Shibboleth Project, http://shibboleth.internet2.edu/

Acknowledgements

- Shawn Bayern and Drew Mazurek, for their great work on CAS.
- The ESUP-Portail SSO group for their feedback and contributions.

Vitae

- **Pascal Aubry** formerly played with real-time systems at ECP until 1993. In successive years he worked at IRISA on the distribution of synchronous programs and received his Ph.D. degree in Computer Science in 1997. Currently at IFSIC, University of Rennes 1, he manages web-projects. He is part of the ESUP-Portail project since its beginning in late 2002, involved in web security (SSO, authorizations) and data storage.
- **Vincent Mathieu** is in charge of network deployment and administration at University of Nancy 2. LDAP expert, he also manages some internet services. He is the leader of the ESUP-Portail SSO group.
- **Julien Marchal** is in charge of email services and some other network-related web applications at University of Nancy 2. He is also part of the ESUP-Portail group, involved in SSO, and communication services, and leader of the ESUP-Portail uPortal group.