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

Pascal Aubry[1], Vincent Mathieu[2], Julien Marchal[2]
[1]IFSIC, University of Rennes 1, France,
pascal.aubry@univ-rennes1.fr
[2]University of Nancy 2, France,
vincent.mathieu@univ-nancy2.fr, julien.marchal@univ-nancy2.fr

Abstract

The universality of the HTTP protocol seduced developers for quite a long time; in fact, today, most applications are web-based. LDAP directories may make life easier on our users' brains by making them memorize only one password, but their fingers are still very busy with all the authentications they need to type — in practice, each time they access an application.

Many solutions for Single Sign-On (SSO) are already available. In this article, we describe a free, simple, complete and reliable solution: the CAS (Central Authentication Service), developed by Yale University (New Haven, CT; USA). CAS has been chosen by the French ESUP-Portail consortium, which provides a complete and open solution to Universities and University-level colleges who wish to offer integrated access to their services and information for their students and staff.
Keywords: Single Sign-On (SSO), open-source, authentication.

Povzetek

ESUP-Portail: odprtokodna rešitev problema enkratne prijave z uporabo CAS

Univerzalnost protokola HTTP že dolgo privlači razvijalce tako, da danes do večine aplikacij dostopamo po svetovnem spletu. Direktoriji vrste LDAP nam prihranijo miselni napor, ker omogočajo, da si je potrebno zapomniti le eno geslo, vendar imajo prsti še vedno "polne roke dela", ker je potrebno geslo vnašati večkrat – v praksi vsakič, ko dostopamo do neke aplikacije.

Za enkratno prijavo (Single Sign-On ali SSO) obstaja več rešitev. V pričujočem članku opišemo odprtokodno, enostavno, popolno in zanesljivo rešitev, ki se ji pravi CAS (Central Authentication Service), razvita pa je bila na univerzi Yale (New Haven, Connecticutt, ZDA). CAS uporablja tudi francoski konzorcij ESUP-Portail, ki ponuja odprtokodno in popolno rešitev naloge zagotavljanja integriranega dostopa do storitev in informacij na univerzah in visokih šolah s strani študentov in zaposlenih.
Ključne besede: enkratna prijava, odprta koda, avtentikacija.

1 Why do we need Single Sign-On?

Web-based applications (mailers, forums, agendas, other specific applications) have spread widely over our networks during recent years. These applications often require authentication.

Using LDAP directories provides users with a single account, which is obviously a real improvement. However, some issues remain:

- **Multiple authentications**: it is still necessary to give a netId/password for each application;
- **Security**: because user accounts are unique, password stealing is really a critical problem; for this reason, the security of the authentication process is essential. In addition, security concerns indicate that user credentials should no longer be given to applications.
- **Several authentication mechanisms**: there are a variety of authentication mechanisms. LDAP is one widely used standard today, but it can be replaced by other user databases. In addition, some users possess personal X509 certificates [1[, which can be used for authentication. In any case, bypassing authentication mechanisms would be interesting because it would permit the use of mixed authentication, for instance.
- **Cooperation**: particularly in the educational community, close institutions would like to be able to share resources and applications. Such cooperation implies that users should be identified by an establishment when only authenticated with another one.

- **Authorization**: applications often need to know user profiles to allow (or deny) the performance of specific actions.

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

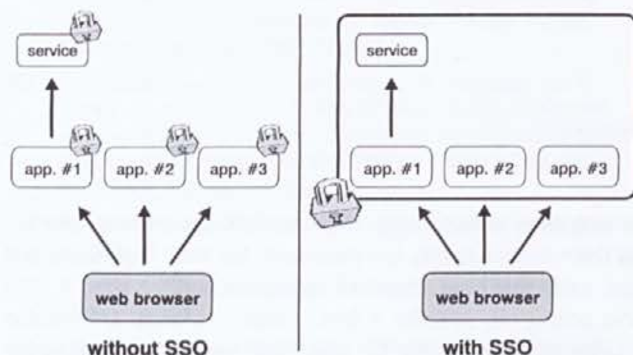

Figure 1: The principle of Single Sign-On

Most SSO mechanisms [2] try to achieve this goal using similar techniques:

- **Authentication is centralized** via an encrypted tunnel in a unique server, which is the only machine to receive user credentials;
- **HTTP redirections** are sent from applications to the authentication server for unauthenticated users, and then back to the applications once a user is authenticated;
- **Information is passed** by the authentication server to applications during the redirections, via 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]. However, because the ESUP-Portail project is based on open-source software only, neither of these commercial solutions was a feasible choice for this project (however, none of them was fitting to our needs). After testing several free implementations, 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

We chose for a number of reasons. CAS is made up of java servlets, can be run on any (JSP spec 1.2 compliant) servlet engine, and offers a web-based authentication service. Its strong points include security, and flexibility, as well as its proxying features and numerous client libraries.

2.1 Security

Security is insured the following ways:

- Passwords are always sent through an encrypted tunnel, and pass only from the browser to the authentication server (Figure 2);
- 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 learn user identities from 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). Thus, as it is the case for almost all serious SSO mechanisms, applications never see any password.

2.2 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 that provides several connectors (LDAP, X509 certificates, NIS domains, databases). These connectors can be used separately, or they can be used in combination to permit mixed authentication. This generic

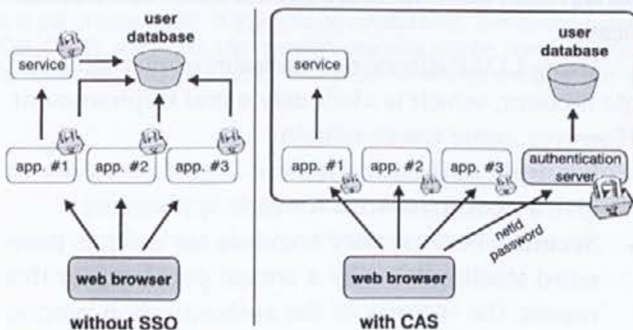

Figure 2: **A software environment secured by CAS**

handler can also been extended to allow system administrators access to other authentication methods, such as Kerberos or Active Directory.

2.3 Authentication proxying

Classic SSO mechanisms demand direct communication between the browser and the application, which prohibits the use of multi-tier installations in which an application must request a back-end service requiring authentication (for instance, a portal requesting a web service).

CAS v2.0 solves this problem by providing an elegant way to propagate authentication without propagating passwords; dedicated tickets (PGT: Proxy Granting Ticket and PT: Proxy Ticket) ensure the validity of user identities to third-party applications. This feature is obviously the strongest point of CAS, inherited from Kerberos concepts.

2.4 Client libraries

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

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

A PAM (**Pluggable Authentication Module** [7[) module (pam_cas) allows non web-based applications to be integrated at a very low level.

2.5 Moreover...

In addition to the reasons described above, CAS has another overwhelming argument in its favor: it is used by many American Universities, in conjunction with LDAP or Kerberos-based authentication, making us confident of its **durability** over time.

Even better, CAS can be plugged directly into **uPortal** [8], (used in the ESUP-Portail) which is on the way to becoming 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 single machine, called the CAS server. This machine is the only actor that knows user passwords. It has a double role:

- Authenticating users;
- Transmitting and certifying the identities of authenticated users to CAS clients.

3.1.2 Web browsers

Web browsers must meet the following requirements to take advantage of all CAS's easy features. They must:

- Own an **encryption engine** so that it can 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 Š3Ć. In particular, for security purposes, private cookies should be transmitted only to the machines that emitted them.

These requirements are met by all classic web browsers, such as 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 include:

- **Libraries**, compatible with 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 operating procedure

3.2.1 User authentication

A previously non-authenticated user (or a user whose authentication has expired) accessing the CAS server

is presented with an authentication form, on which (s)he is invited to enter a netId and a password (Figure 3).

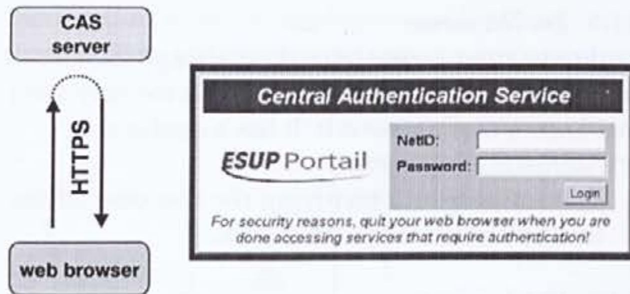

Figure 3: **First access of a browser to the CAS server**

If the netId and password are correct, the server sends a cookie called TGC (Ticket Granting Cookie) to the browser (Figure 4).

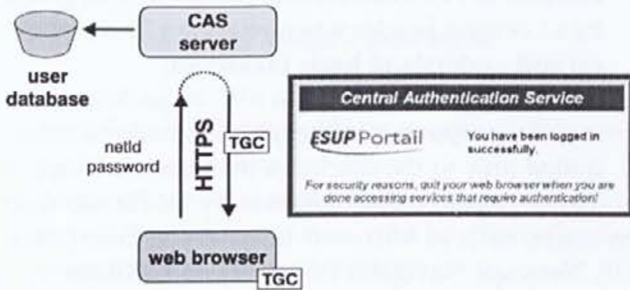

Figure 4: **Authentication of a browser against the CAS server**

This TGC is the user's passport to the CAS server. Its validity is limited, typically to a few hours. The TGC allows web browsers to get CAS client tickets from the CAS server, without needing to re-authenticate. A private cookie that is transmitted only to the CAS server, it is also protected ensuring that all requests to the CAS server are secured. Like all CAS tickets, it is opaque, containing no information about the user. It is simply a session identifier operating between the web browser and the CAS server.

3.2.2 Accessing protected web resources after authentication

When accessing a resource protected by a CAS client, the web browser is redirected to the CAS server. The browser, if previously authenticated, presents its TGC to the CAS server (Figure 5).

On presentation of the TGC, the CAS server delivers a Service Ticket (ST), an opaque ticket providing no user information, that is usable only by the service that asks for it. At the same time, the CAS server re-

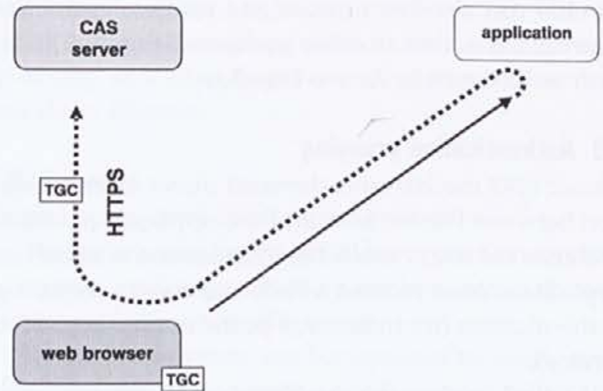

Figure 5: **Redirection of an unknown browser to the CAS server**

directs the browser to the calling service (the Service Ticket is a CGI parameter) (Figure 6).

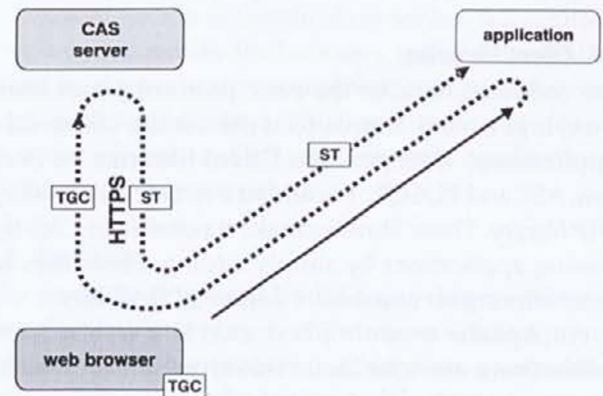

Figure 6: **Redirection of the browser to the calling service after authentication**

The ST is then validated by the CAS client with the CAS server (via an HTTP request) and the desired resource can be delivered to the browser (Figure 7).

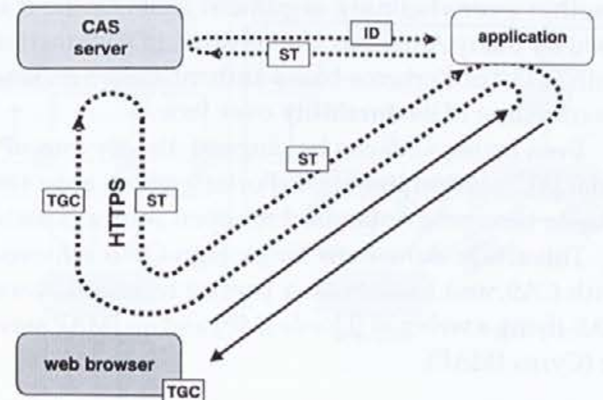

Figure 7: **Validation of a Service Ticket**

All the redirections mentioned above are transparent for the user: (s)he accesses the resource without being authenticated, without any interaction at all (Figure 8).

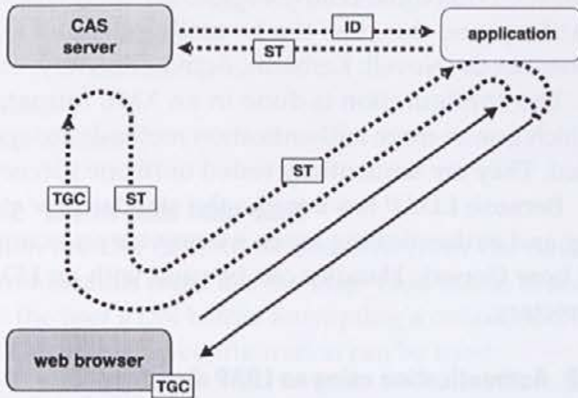

Figure 8: **User view of the CAS redirections**

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

3.2.3 Accessing protected web resources without authentication

The non-authenticated web browser attempting to access a resource protected by a CAS client is still redirected to the CAS server. As no TGC is given, the CAS server returns an authentication form.

Once the browser has submitted the form and is correctly authenticated, the CAS server:

- Sends the browser a TGC, that will exempt it from re-authenticating later;
- Provides a Service Ticket and redirects the browser to the CAS client calling service.

Thus, there is no need to be previously authenticated to access a protected resource: authentication is automatically performed the first time a user requests a protected resource.

3.3 Multi-tier installations

3.3.1 CAS proxies

The CAS multi-tier feature makes it possible for a CAS client to access a back-end service under the authenticated user's identity. A CAS client that is able to proxy credentials is called a CAS proxy. Most used CAS proxies are:

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

In a multi-tier CAS installation, CAS clients no longer have access to the browser's cookie cache, and so redirections can not be used.

3.3.2 2-tier installations

A CAS proxy, when validating a Service Ticket to authenticate a user, also asks for a PGT (Proxy Granting Ticket) from the CAS server (Figure 9).

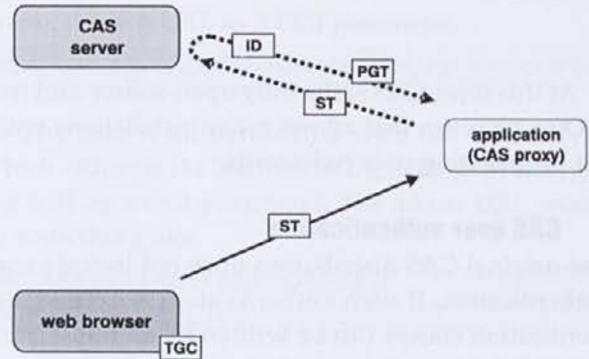

Figure 9: **PGT retrieval by a CAS proxy**

A PGT is the CAS proxy's passport, representing a user, to the CAS server. It allows the CAS proxies to obtain tickets for CAS back-end services from the CAS server, without needing to validate an ST. The PGT is an opaque and re-playable ticket, delivered by the CAS server in response to a secured request, to insure TGC integrity and confidentiality. PGT validity is limited, like that of TGC, to a few hours.

PGT are to applications what TGC are to web browsers. A PGT allows applications (CAS proxies) to authenticate a user with the CAS server, and to obtain Proxy Tickets (PT are to CAS proxies what ST are to web browsers). Proxy Tickets, like Service Tickets, are validated by the CAS server before allowing access to protected resources (Figure 10).

3.3.3 N-tier installations

Obviously, the back-end service accessed by the CAS proxy in 2-tier installation can itself be a CAS proxy. CAS proxies can be chained, as shown in Figure 11.

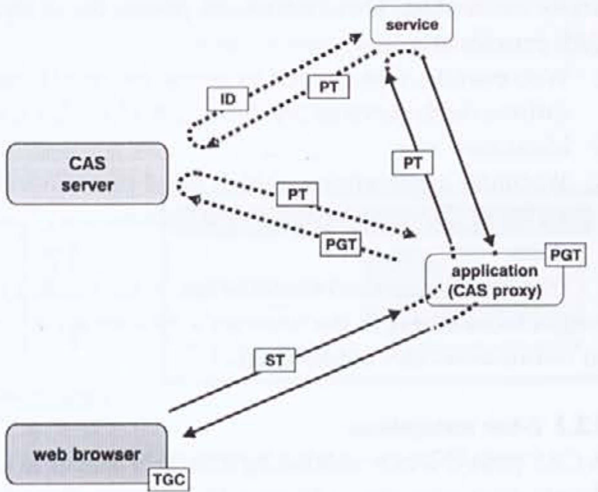

Figure 10: **Validation of a Proxy Ticket by a back-end service**

At this time, CAS is the only open-source and free SSO mechanism that allows n-tier installations without propagating user passwords.

4 CAS user authentication

The original CAS distribution does not include user authentication. If such authentication is needed, authentication classes can be written by administrators to fulfill their exact needs. Some example classes are provided by the cas-server distribution, for test purposes only.

4.1 The GenericHandler class

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

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

Because LDAP has become the standard for storing and authenticating users, we provide an example of how Generic Handler can be used with an LDAP directory.

4.2 Authentication using 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 with LDAP directories in which user DN (Distinguished Name) can be directly deduced from their netId. In practice, these are directories in which 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.

Figure 11: **A chain of CAS proxies**

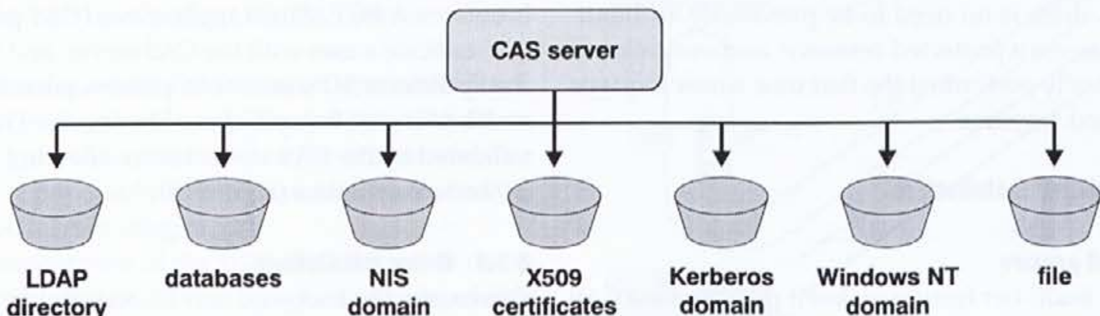

Figure 12: **User authentication with ESUP-Portail GenericHandler**

The following configuration can be used:

```
<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 netId, administrators must use the ldap_bind mode to seek out the user's DN before attempting a connection.

The following configuration can be used:

```
<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 permits redundancy in order 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 kinds of CAS applications exist:

- **CAS "simple" clients** only need to authenticate users.
- **CAS proxies** need to both authenticate users, and use tier services. They need to be able to retrieve PGT from the CAS server, and later to transmit PT to back-end services in order to authenticate the users for whom they act.
- **CAS back-end services** need to validate the PT given by CAS proxies and to obtain user identities.

5.1 "simple" CAS clients

The principle is to use a function (or method) that 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 redirection as shown in 3.2.2);
- 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 – in our case, phpCAS [12] – 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 is authenticated with the CAS server, which redirects the browser to the calling service, giving a ST as a CGI parameter. The access URL would be something like:

```
http://test.univ.fr/script.php?ticket=ST-2
uw2KEWinSFeZ9fotZlio
```

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

```
http(s)://auth.univ.fr/serviceValidate?service=http://
test.univ.fr/script.php&ticket=ST-2-uw2KEWinSFeZ9fotZlio
```

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 with the 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 with 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 one way that 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 begins exactly as the one 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); and
- A PGT, using the callback URL.

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

Java and PHP libraries mask the complexity of all this when developing a CAS proxy. The following is an example of the way a CAS proxy can be implemented using 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 with the CAS server (instead of a Service Ticket).

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

For obvious performance reasons, applications should maintain sessions so that the CAS mechanism is fired only once, rather than for each request.

This remark is true for CAS clients and proxies, which should maintain a session with the browser, as well as for back-end services, which should maintain a session with the CAS proxy.

5.4.2 Asynchronism

When using CAS client libraries, retrieving a PGT for a user in a CAS proxy is easy. However, developers should be aware of the possibility of desynchronization between the different sessions of a multi-tier CAS installation.

Imagine that a user connects to a web portal, which will act as a CAS proxy. The user is authenticated with 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 obtain the new PTs needed to access back-end services.

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

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, access to a site (or part of it) can require an authentication from 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 single authentication service for web applications that is both simple and efficient. CAS offers more by allowing the CAS-ification of non-web services, such as IMAP, FTP, etc.

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

6.1 The PAM pam_cas module

Pam_cas is included in CAS client distribution. Though it is powerful, it is also light (about 300 lines of C, half of them shared with mod_cas).

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

Pam_cas can not be used outside of a multi-tier installation: the CAS-ified service must be accessed by a CAS proxy. Indeed, it is inconceivable for a human being (a human user of an FTP service, for instance) to provide a CAS ticket.

Fortunately, the PAM modular concept allows pam_cas to be used in conjunction with other PAM modules. It is possible for a service to authenticate a user both using a traditional method (a netId and a password) and using a CAS method (netId and ticket).

The example below shows how this can be done.

6.2 Using pam_cas to CAS-ify an IMAP server

Our goal here is to CAS-ify an IMAP server in order to accept connections from a web portal using Proxy Tickets, while continuing to accept connections from traditional mail clients using passwords.

If the IMAP server is PAM-compliant (which is generally the case), the PAM configuration can be, for instance:

```
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 attempted in three different ways: with an LDAP directory, with the local Unix user database, and eventually with pam_cas. The secret provided is validated with the CAS server (internally, only if it is ticket-shaped for obvious performance reasons).

6.3 CAS-ifying the Cyrus-IMAP server

The IMAP protocol is very specific, and is probably the most difficult to CAS-ify. IMAP clients and webmails in general have the odd habit of generating large numbers of requests, by closing and opening connections repeatedly. This, of course, leads to numerous authentication requests for the CAS server.

When using a traditional webmail (where users authenticate with their netId and password), the only consequence is a heavier load for the web server running the webmail. However, within a CAS multi-tier installation, the load increase is supported both by the web server running the webmail, and by the CAS server.

Asking for and validating tickets for each request is clearly prohibitive from the point of view of performance. Consequently, a cache is needed on the IMAP server to allow the webmail to re-play the PT.

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

This daemon, which communicates with Cyrus-SASL via a Unix socket, proposes a cache mechanism. This cache allows the mail client 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 reduced authentication requests by 95%. Only 5% were really played, i.e. with the tickets being validated by the CAS server.

6.4 CAS-ifying Horde IMP

Our primary goal was to add a webmail product into the ESUP-Portail software, if possible completely integrated into our SSO. We decided to try Horde IMP [13].

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

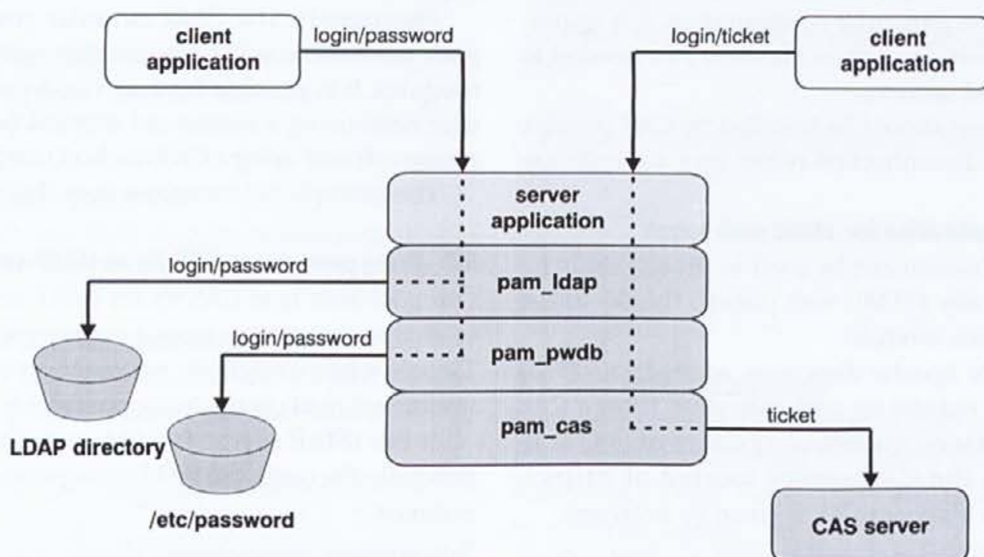

Figure 13: **Using pam_cas**

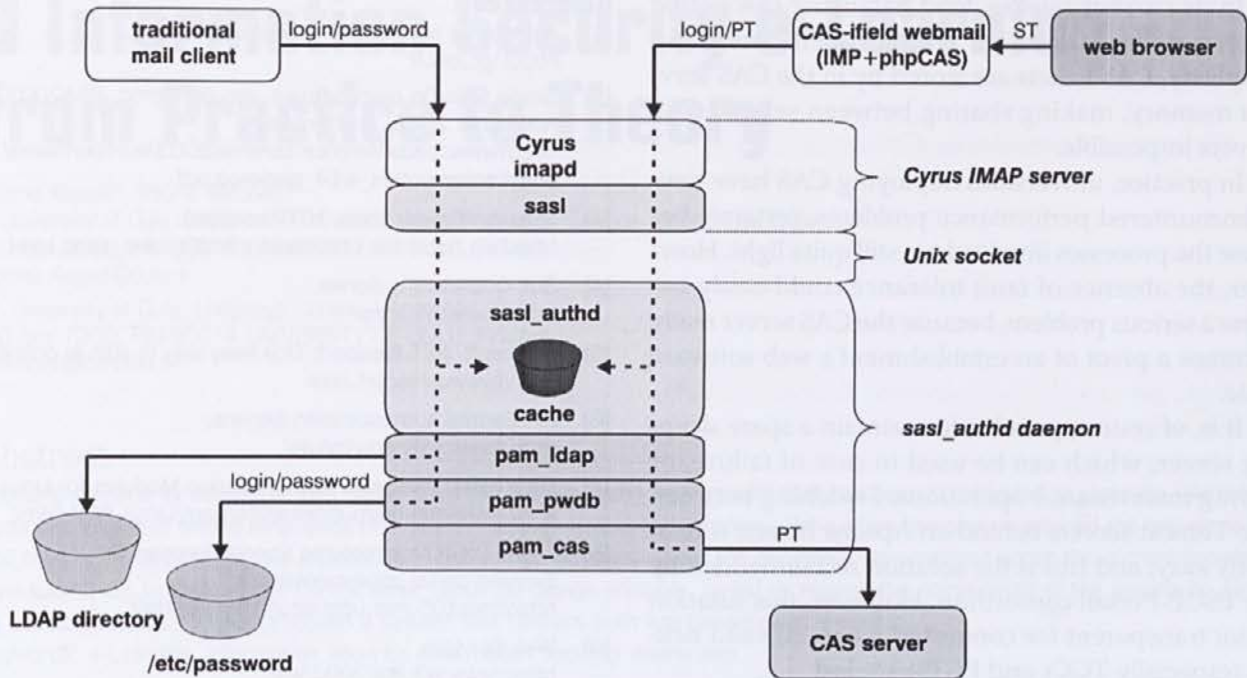

Figure 14: **CAS-ification of Cyrus-IMAP**

way that passwords are, although their validity is limited. In other words, the webmail can use a PT several times thanks to the IMAP cache, but a PT stored in the IMAP cache can be erased (because of the garbage collector of the IMAP cache), supplanted 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 accesses a traditional mail client. In all of these situations, the next connection with the PT would be refused by the IMAP server. To get around this problem, the webmail was modified to allow a new PT to be acquired from the CAS server, in order to make a second attempt at an IMAP connection.

Obviously, using CAS client libraries are not as simple as was implied in 5.1.1 ("Writing a PHP CAS client").

7 Restrictions and perspectives

In the first 6 sections of this article, we have described the strong points of CAS:

- CAS is an open-source, free product;
- CAS offers a more than satisfactory level of security;
- A CAS server is very easy to set up and configure;
- Web applications are very easy to CAS-ify.

In this section, we will examine its limitations, and offer some ideas for getting around these weak points.

7.1 CAS brings SSO, nothing else

CAS is a Single Sign-On mechanism that can also run at the system-level, thanks to pam_cas. On the other hand, it is strictly limited to user authentication: it does not (and probably never will) deal with authorizations or with the propagation of user attributes.

Moreover, the user databases are local, at the establishment-level. CAS does not address multi-establishments issues. Recent developments on Sympa [14] show an elegant way to allow users from several establishments to be authenticated, by relying on several CAS servers. However, the most promising way to permit different establishments to cooperate using 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, for reasons of efficiency and simplicity, CAS tickets are stored by in the CAS server's memory, making sharing between several CAS servers impossible.

In practice, universities deploying CAS have never encountered performance problems, certainly because the processes involved are still quite light. However, the absence of fault tolerance could easily become a serious problem, because the CAS server really becomes a pivot of an establishment's web software suite.

It is, of course, possible to maintain a spare sleeping server, which can be used in case of failure, or during maintenance operations. Switching between two Tomcat servers behind an Apache frontal is relatively easy, and this is the 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.

Another possibility is to store granting tickets (TGCs and PGTs) in a database. This is conceivable since in this case, switching from one CAS server to another would have a very limited effect on tickets (only STs and PTs would be lost), while preserving simplicity and thus performance.

8 What about CAS in the future?

The ESUP-Portail consortium has taken 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.

We have every confidence in CAS. CAS has been adopted by the ESUP-Portail consortium as its SSO software, and will be deployed in the coming months in all those French Universities who choose the ESUP-Portail software. We strongly believe that it can become a standard.

References

[1] Autorité de certification du CRU, in French, http://igc.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.

Pascal Aubry played with real-time systems at ECP until 1993. In the succeeding years, he worked at IRISA on the distribution of synchronous programs and received his Ph.D. in Computer Science in 1997. Currently at IFSIC, University of Rennes 1, he manages web-projects. He has been part of the ESUP-Portail project since its beginning in late 2002, working on web security (SSO, authorizations) and data storage.

Vincent Mathieu is in charge of network deployment and administration at University of Nancy 2. An 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 other network-related web applications at University of Nancy 2. He is also part of the ESUP-Portail group, working on SSO and communication services, and is the leader of the ESUP-Portail uPortal group.