Single Sign-On open-source avec CAS (Central Authentication Service)

Vincent Mathieu
Université de Nancy 2
vincent.mathieu@univ-nancy2.fr

Pascal Aubry
IFSIC - Université de Rennes 1
pascal.aubry@univ-rennes1.fr

Julien Marchal
Université de Nancy 2
julien.marchal@univ-nancy2.fr

Date : 12 Octobre 2003

Résumé

L'universalité du protocole HTTP a depuis longtemps séduit les développeurs ; les applications portées sur le web sont de plus en plus nombreuses.
La mise en place d'annuaires (LDAP par exemple) a épargné la tête des utilisateurs en ne leur faisant mémoriser qu'un seul mot de passe, mais leurs doigts sont encore durement sollicités car ils doivent s'authentifier chaque fois qu'il accèdent une application.
Plusieurs solutions de Single Sign-On (authentification unique et unifiée) sont d'ores et déjà disponibles dans le commerce. Cet article décrit une solution libre, simple, riche et sûre : CAS (Central Authentication Service), développée par l'Université de Yale, et adoptée par le projet ESUP-Portail.

Mots clefs

Single Sign-On, open-source, web, sécurité, authentification.

1 Pourquoi le Single Sign-On ?

Les services numériques accessibles par le *web* (*intranet*, courrier électronique, forums, agendas, applications spécifiques) à disposition des étudiants, enseignants, personnels administratifs se sont multipliés en quelques années. Ces services nécessitent très souvent une authentification.

L'utilisation de techniques de synchronisation entre domaines d'authentification hétérogènes, puis de serveurs LDAP a permis la mise en oeuvre d'un compte unique (*login* / mot de passe) pour chaque utilisateur, ce qui est un progrès. Se posent maintenant les problèmes suivants :

- **authentifications multiples** : il est nécessaire d'entrer son *login*/mot de passe lors de l'accès à chaque application.
- **sécurité** : le compte étant unique, le vol de celui-ci entraîne un risque très important. La sécurisation de l'authentification devient donc primordiale. Il est également fortement souhaitable que les applications n'aient pas connaissance du mot de passe.
- **différents mécanismes d'authentification** : certains utilisateurs disposent de certificats X509 [1][2], qui pourraient servir à l'authentification. En outre, il n'est pas exclu que l'utilisation de LDAP à cette fin ne soit pas remplacée à terme par autre chose, et que certaines politiques d'établissement exigent l'utilisation de bases de données additionnelles. Il semble donc intéressant de disposer d'un service d'abstraction par rapport au(x) mécanisme(s) d'authentification local(aux).
- **aspects multi-établissements** : le compte d'un utilisateur est unique à l'intérieur de l'établissement ; il serait souhaitable que l'accès à des ressources informatiques d'un autre établissement puisse se faire à l'aide du même compte.
- **autorisations** : il est nécessaire pour certaines applications de pouvoir disposer d'informations définissant les rôles des utilisateurs.

Les mécanismes de **SSO** (**S**ingle **S**ign-**O**n : authentification unique, et une seule fois) [3] tentent de répondre à ces problématiques, en utilisant tous des techniques assez semblables, à savoir :

- une **centralisation de l'authentification** sur un serveur qui est le seul à recueillir les mots de passe des utilisateurs, à travers un canal chiffré ;
- des **redirections HTTP** transparentes du navigateur client, depuis les applications vers le serveur d'authentification, puis du serveur vers les applications.
- le **passage d'informations entre le serveur d'authentification et les applications** à l'aide de *cookies* [4], et/ou de paramètres CGI de requêtes HTTP (GET ou POST).

Parmi les différentes solutions commerciales offertes aux administrateurs et développeurs émergent *Sun ONE Identity Server* [5] et *MicroSoft Passport* [6]. Cet article se propose de montrer qu'une solution libre permet d'implémenter un mécanisme de SSO puissant, sûr et souple pour les applications *web*.

2 Le choix de CAS

Développé par l'Université de Yale, CAS (*Central Authentication Service* [7]) met en oeuvre un serveur d'authentification accessible par W3, composé de servlets java, qui fonctionne sur tout moteur de *servlets* (*Tomcat* par exemple), et dont les points forts sont listés ci-dessous.

- La **sécurité** est assurée par les dispositifs suivants :
  - le mot de passe de l'utilisateur ne circule qu'entre le navigateur client et le serveur d'authentification, nécessairement à travers un canal crypté.
  - Les ré-authentifications suivantes sont faites de manière transparente à l'utilisateur, sous réserve de l'acceptation d'un *cookie* privé et protégé. Seul le serveur d'authentification peut lire et écrire ce *cookie*, qui ne contient qu'un identifiant de session.
  - L'application reçoit du serveur d'authentification un « ticket opaque » qui n'est pas porteur d'information personnelle. Ce ticket circule en clair via le navigateur (en paramètre CGI) ; il n'est pas rejouable, a une durée de vie courte, est n'est utilisable que par l'application qui l'a demandé. L'application va ensuite contacter directement (en http) le serveur CAS afin de faire valider (et expirer) ce ticket ; le serveur CAS va retourner à l'application l'identifiant de la personne, validé. L'application n'a ainsi jamais accès au mot de passe (schéma pourtant classique de pratiquement tous les mécanismes de SSO).
- Les mécanismes classiques imposent une communication entre le navigateur web et l'application, ce qui exclut les **configurations n-tiers**, où une application doit directement interroger un service nécessitant authentification (c'est le cas par exemple pour un portail accédant à un *web service*). CAS, dans sa version 2.0, résout ce problème en proposant un mécanisme de mandataires (*proxies*). Des tickets dédiés permettent à des applications tierces, n'ayant aucune communication avec le navigateur client, d'être assurées de l'authentification de l'utilisateur. Cette fonctionnalité est assurément le point fort de CAS.
- Le package proposé implémente tout le protocole de mise en oeuvre du SSO, à l'exception du module d'authentification locale qui est à la charge de l'administrateur du serveur d'authentification. Cela laisse la **liberté d'implémenter exactement l'authentification souhaitée** (LDAP, *Kerberos* [8], certificats X509, NIS, un panachage, ...).
- Des **librairies clientes** en *Java*, *Perl*, JSP, ASP, PL/SQL et PHP sont livrées. Cela permet une grande souplesse sur les serveurs d'applications. L'intégration dans des outils utilisés dans le monde universitaire est d'ores et déjà faite, comme celle d'*uPortal* [9].
- L'utilisation de *cookies* exclusivement privés dans CAS (passage de tickets entre serveur d'authentification et applications uniquement sous forme de paramètres de GET HTTP) permet à CAS d'être opérationnel sur des serveurs situés dans des **domaines DNS différents**.
- Un **module Apache** (*mod_cas*) permet d'utiliser CAS pour protéger l'accès à des documents *web* statiques, les librairies clientes ne pouvant être utilisées dans ce cas.
- Un **module PAM** [10] (*pam_cas*) permet de « CAS-ifier » des services non *web*, tels que FTP, IMAP, ...
- Enfin, CAS est en production dans plusieurs Universités américaines, avec des authentifications internes *Kerberos* ou LDAP, ce qui permet d'être confiant sur sa **fiabilité**[1].

Nous vous proposons de détailler dans cet article le fonctionnement d'une implémentation de SSO avec CAS, en nous attachant à présenter les avantages et inconvénients de cette solution.

3 Le mécanisme CAS

3.1 Architecture

3.1.1 Le serveur CAS

L'authentification est centralisée sur une machine unique, le serveur CAS. Ce serveur est le seul acteur du mécanisme CAS à avoir connaissance des mots de passe des utilisateurs. Son rôle est double :
- **authentifier** les utilisateurs ;
- **transmettre et certifier l'identité** de la personne authentifiée (aux clients CAS).

*3.1.2 Les navigateurs (*web*)*

Les navigateurs doivent satisfaire les contraintes suivantes pour bénéficier de tout le confort de CAS[2] :

[1] À l'Université d'Indiana, CAS contrôle environ 80 applications web destinées à une centaine de milliers d'utilisateurs ; Le serveur CAS y effectue une moyenne de 9000 authentifications par jour.

- disposer d'un moteur de **chiffrement** leur permettant d'utiliser le protocole HTTPS ;
- savoir effectuer des **redirections HTTP** (accéder à une page donnée dans une entête Location lors d'une réponse 30x à une première requête HTTP) et interpréter le langage ***JavaScript*** ;
- savoir stocker des ***cookies***, comme défini par [4]. En particulier, les *cookies* privés ne devront être retransmis qu'au serveur les ayant émis pour garantir la sécurité du mécanisme CAS.

Ces exigences sont satisfaites par tous les navigateurs classiquement utilisés, à savoir *MicroSoft Internet Explorer* (depuis 5.0), *Netscape Navigator* (depuis 4.7) et *Mozilla*.

3.1.3 Les clients CAS

Une application web muni d'une librairie cliente ou un serveur web utilisant le module *mod_cas* est alors appelé « client CAS ». Il ne délivre les ressources qu'après s'être assuré que le navigateur qui l'accède se soit authentifié auprès du serveur CAS.

Parmi les clients CAS, on trouve :
- des librairies correspondant aux langages communément employés en programmation web dynamique (*Perl*, *Java*, JSP, PHP, ASP) ;
- un module *Apache*, qui permet de protéger des documents statiques ;
- un module PAM, qui permet d'authentifier les utilisateurs au niveau système.

3.2 Fonctionnement de base

3.2.1 Authentification d'un utilisateur

Un utilisateur non déjà précédemment authentifié, ou dont l'authentification a expiré, et qui accède au serveur CAS se voit proposer un formulaire d'authentification, dans lequel il est invité à entrer son nom de connexion et son mot de passe :

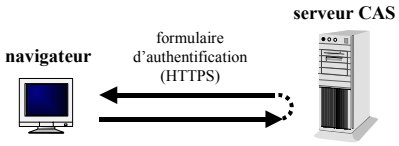

Figure 1 : *Premier accès d'un navigateur au serveur CAS (sans TGC)*

Si les informations sont correctes, le serveur renvoie au navigateur un *cookie* appelé TGC (*Ticket Granting Cookie*) :

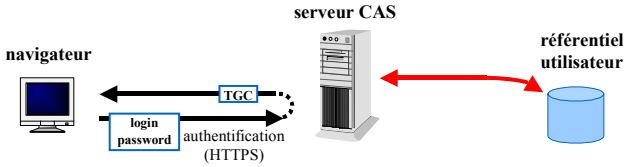

Figure 2 : *Authentification d'un navigateur auprès du serveur CAS*

Le *Ticket Granting Cookie* (TGC) est le passeport de l'utilisateur auprès du serveur CAS. Le TGC, à durée de vie limitée (typiquement quelques heures), est le moyen pour les navigateurs d'obtenir auprès du serveur CAS des tickets pour les clients CAS sans avoir à se ré-authentifier. C'est un *cookie* privé (n'est jamais transmis à d'autres serveurs que le serveur CAS) et protégé (toutes les requêtes des navigateurs vers le serveur CAS se font sous HTTPS). Comme tous les tickets utilisés dans le mécanisme CAS, il est opaque (ne contient aucune information sur l'utilisateur authentifié) : c'est un identifiant de session entre le navigateur et le serveur CAS.

3.2.2 Accès à une ressource protégée après authentification

Lors de l'accès à une ressource protégée par un client CAS, celui-ci redirige le navigateur vers le serveur CAS dans le but d'authentifier l'utilisateur (cf Figure 3). Le navigateur, précédemment authentifié auprès du serveur CAS, lui présente le TGC (rappel : un *cookie*).

---

[2] Si le moteur de chiffrement est absolument indispensable pour utiliser CAS, les autres contraintes, si elles ne sont pas satisfaites, n'empêchent pas le mécanisme CAS de fonctionner. Un navigateur ne sachant pas effectuer les redirections ou n'interprétant pas le langage JavaScript forcera l'utilisateur à effectuer un clic à chaque redirection (qui ne sera alors plus transparente). Un navigateur ne stockant pas les cookies forcera l'utilisateur à entrer ses informations d'authentification à chaque passage par le serveur d'authentification.

Sur présentation du TGC, le serveur CAS délivre au navigateur un *Service Ticket* (ticket de service ou ST) ; c'est un ticket opaque, qui ne transporte aucune information personnelle ; il n'est utilisable que par le « service » (l'URL) qui en a fait la demande. Dans le même temps, il le redirige vers le service demandeur en passant le *Service Ticket* en paramètre CGI (cf Figure 4).

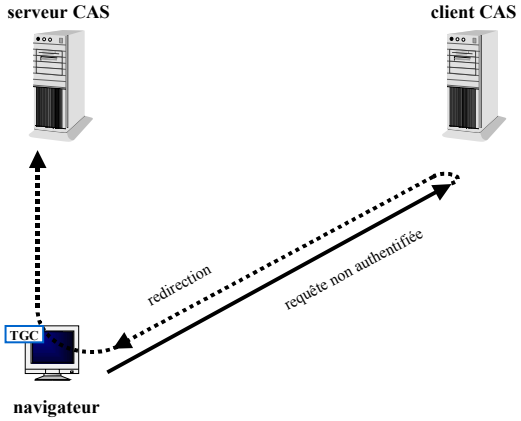

Figure 3 : *Redirection vers le serveurCAS d'un navigateur non authentifié auprès du client CAS*

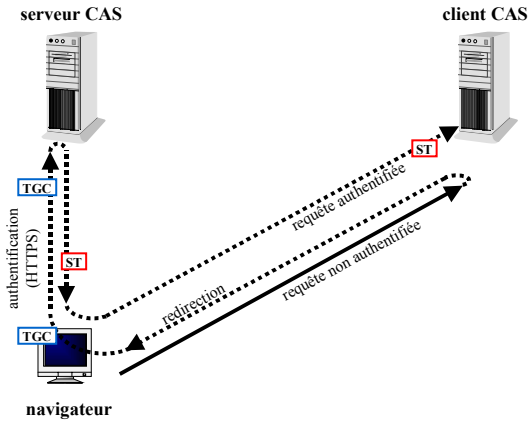

Figure 4 : *Redirection par le serveur CAS d'un navigateur vers un client CAS après authentification*

Le *Service Ticket* est alors validé auprès du serveur CAS par le client CAS, **directement en http**, et la ressource peut être délivrée au navigateur (cf Figure 5).

Il est à noter que toutes les redirections présentées Figure 5 sont complètement transparentes pour l'utilisateur. Celui-ci ne voit pas les redirections et a l'impression d'accéder directement à la ressource désirée, sans authentification (cf Figure 6).

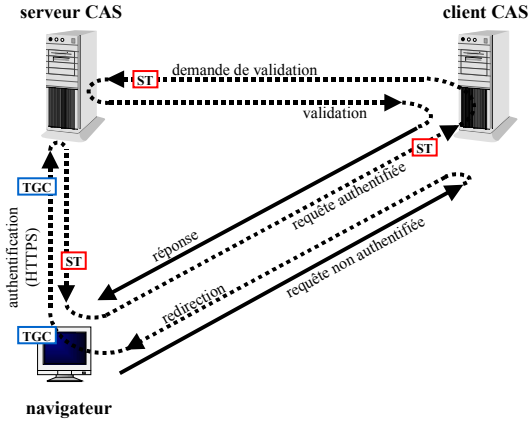

Figure 5 : *Validation d'un Service Ticket par un client CAS auprès du serveur CAS*

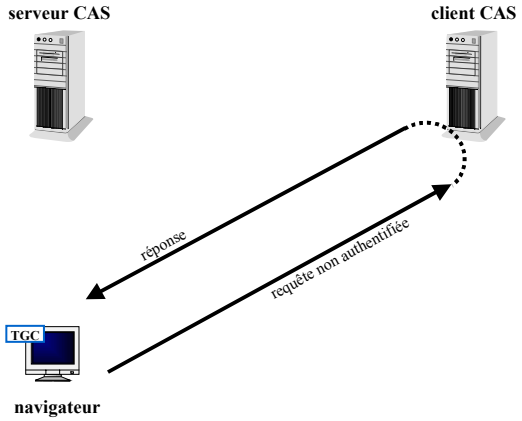

Figure 6 : *Vision par l'utilisateur du mécanisme d'authentification*

Le *Service Ticket* (ST) est le passeport de l'utilisateur auprès d'un client CAS. Il est non rejouable (ne peut être présenté qu'une seule fois au serveur CAS), limité à un seul client CAS et sa durée de vie est très limitée dans le temps (typiquement quelques secondes).

3.2.3 Accès à une ressource protégée sans authentification préalable

Si l'utilisateur n'est pas déjà authentifié auprès du serveur CAS avant d'accéder à une ressource, son navigateur est, comme précédemment, redirigé vers le serveur CAS, qui lui propose alors un formulaire d'authentification.

Lors de la soumission du formulaire par le navigateur au serveur CAS, si les informations fournies sont correctes, le serveur CAS :

- délivre un TGC (*Ticket Granting Cookie*) au client, qui lui permettra ultérieurement de ne pas avoir à se ré-authentifier ;

- délivre au client un *Service Ticket* à destination du client CAS ;
- redirige le client vers le client CAS.

On voit ainsi qu'il n'est pas nécessaire d'être préalablement authentifié pour accéder à une ressource protégée.

3.3 Fonctionnement multi-tiers

3.3.1 Les mandataires (proxies) CAS

Le fonctionnement n-tiers de CAS consiste en la possibilité, pour un client CAS, de se comporter comme un navigateur. Un tel client CAS est alors appelé mandataire (*proxy*) CAS.

Les exemples d'utilisation de mandataires sont divers :
- un **portail *web***, pour lequel l'utilisateur s'est authentifié, peut avoir besoin d'interroger une application externe sous l'identité de l'utilisateur connecté (un *web service* [11] par exemple) ;
- une **passerelle *web*** de courrier électronique (*webmail*), à laquelle un utilisateur s'est authentifié, a besoin de se connecter à un serveur IMAP pour relever le courrier électronique de l'utilisateur, sous son identité.

Dans le fonctionnement de base, le client CAS est toujours en lien direct avec le navigateur. Dans un fonctionnement n-tiers, le navigateur accède à un client CAS à travers un mandataire CAS. Le mécanisme de redirection vu dans le fonctionnement de base n'est alors plus applicable.

3.3.2 Fonctionnement 2-tiers

Un mandataire CAS, lorsqu'il valide un *Service Ticket* pour authentifier un utilisateur, effectue, dans le même temps, une demande de PGT (*Proxy Granting Ticket*) (cf Figure 7).

Le *Proxy Granting Ticket* (PGT) est le passeport d'un mandataire CAS, pour un utilisateur, auprès du serveur CAS. Le PGT est opaque, rejouable, et obtenu du serveur CAS par un canal chiffré (et un mécanisme de *callback* assurant son intégrité). Comme le TGC, il est à durée de vie limitée (typiquement quelques heures).

Le PGT est l'équivalent, pour les mandataires CAS, des TGCs pour les navigateurs. Il permet d'authentifier l'utilisateur auprès du serveur CAS, et d'obtenir ultérieurement du serveur CAS des *Proxy Tickets*, équivalents pour les mandataires des *Service Tickets* pour les navigateurs.

Les *Proxy Tickets* (PT) sont, comme les *Service Tickets*, validés par le serveur CAS pour donner accès à des ressources protégées (cf Figure 8).

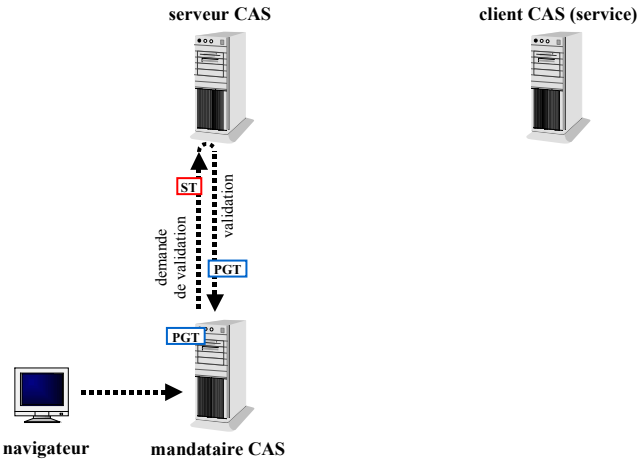

Figure 7 : *Récupération d'un PGT par un mandataire CAS auprès du serveur CAS*

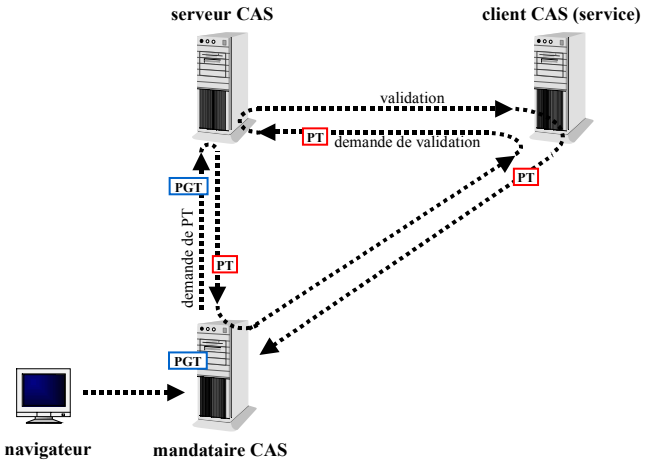

Figure 8 : *Validation d'un PT par un client CAS accédé par un mandataire CAS*

Le *Proxy Ticket* (PT) est le passeport des mandataires CAS auprès des clients CAS. Il est opaque, non rejouable, et à durée de vie très limitée (comme le *Service Ticket*, typiquement quelques secondes).

3.3.3 Fonctionnement n-tiers

On voit aisément que le client CAS accédé par le mandataire CAS du fonctionnement 2-tiers peut également être mandataire à son tour. Les mandataires peuvent ainsi être chaînés (cf Figure 9).
CAS est à notre connaissance, le seul mécanisme de SSO proposant un tel fonctionnement multi-tiers sans aucune propagation de mot de passe.

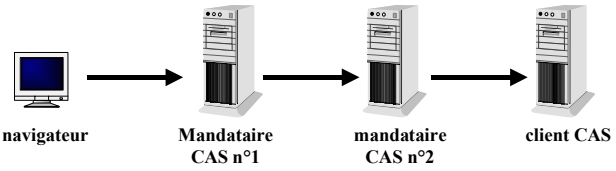

Figure 9 : *Schéma de fonctionnement n-tiers*

4 L'authentification sous CAS

CAS, dans sa distribution originale, ne propose pas de classe d'authentification : les méthodes d'authentification sont à la charge de l'administrateur du serveur CAS[3].

Développée au sein du projet *ESUP-Portail* [12], la classe *GenericHandler* [13] permet l'implémentation de plusieurs méthodes d'authentification plus ou moins traditionnelles : annuaires LDAP, bases de données, domaines NIS, et fichiers. Cette classe peut être aisément étendue à d'autres méthodes d'authentification, selon les besoins des administrateurs de sites (*Novell*, *Kerberos*, *Active Directory*, ...)[4].

La configuration du serveur se fait simplement en renseignant un fichier au format XML. Une ou plusieurs méthodes d'authentification sont spécifiées, et testées séquentiellement jusqu'à authentification de l'utilisateur.

4.1 Authentification sur un annuaire LDAP

Parce que LDAP est devenu un (le) standard de référentiel utilisateur, l'authentification sur un annuaire LDAP est aujourd'hui la méthode la plus souvent utilisée.

Deux modes d'accès à l'annuaire sont proposés, selon la structure interne de l'annuaire.

*4.1.1 Accès direct à l'annuaire (*ldap_fastbind*)*

Le mode *ldap_fastbind* peut être utilisé pour les annuaires dont le DN (*Distinguished Name*) d'un utilisateur peut être directement déduit de son nom de *login*. Pratiquement, il s'agit des annuaires dont tous les utilisateurs sont stockés au même niveau hiérarchique.

CAS tente alors une connexion à l'annuaire sur le DN de l'utilisateur et le mot de passe fourni. L'utilisateur est considéré comme authentifié si la connexion réussit.

On utilisera par exemple :

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

*4.1.2 Recherche dans l'annuaire (*ldap_bind*)*

Lorsque la déduction du DN de l'utilisateur à partir de son nom de connexion est impossible[5], il faut alors utiliser le mode *ldap_bind*, qui effectue une recherche du DN de l'utilisateur dans l'annuaire avant de tenter une connexion.

On utilisera par exemple :

---

[3] Une classe d'authentification basique est fourni avec la distribution cas-server, et d'autres classes étaient disponibles, néanmoins simplistes, ce qui a conduit au développement de *GenericHandler* par le projet *ESUP-Portail*.
[4] Les modules d'authentification s'appuyant sur des certificats X509, ainsi que un domaine *Windows NT*, sont en cours de développement.
[5] Par exemple lorsque les utilisateurs de l'annuaire sont situés dans des hiérarchies différentes.

```
<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>
```

4.1.3 Redondance des annuaires

Afin d'assurer la tolérance aux fautes de l'annuaire LDAP, il est possible de spécifier non pas un seul annuaire LDAP, mais une liste d'annuaires, qui sont alors considérés comme des réplicas. La panne d'un annuaire est alors palliée par la présence de ses réplicas[6].

4.2 Autres modes d'authentification

4.2.1 Sur une base de données

Cette méthode est essentiellement destinée à des organisations dont certains utilisateurs, pour des raisons techniques ou organisationnelles, ne sont pas enregistrés dans leur annuaire LDAP mais dans une base de données[7]. Comme pour l'authentification LDAP, la tolérance aux fautes est assurée par redondance des serveurs de données, et deux modes sont proposés :

- Pour le **mode « connexion »** (*database_bind*), les utilisateurs à authentifier doivent être également utilisateurs déclarés de la base de données. L'authentification est réussie si les informations de connexion fournis par l'utilisateur permettent de se connecter à la base de données.
- Le **mode « recherche »** (*database_search*) utilise une connexion privilégiée à la base, et les informations de connexion des utilisateurs sont stockées dans une table. L'authentification est réussie lorsque qu'une correspondance est établie entre les informations fournies par l'utilisateur et celles contenues dans la base de données.

4.2.2 Sur des certificats personnels x509

Les certificats personnels x509 sont communément utilisés pour l'authentification des applications *web*. Lorsqu'un certificat est transmis par le navigateur au serveur CAS, il peut servir d'authentification pour son utilisateur. Lorsqu'un utilisateur présente un certificat x509, le serveur CAS :

- **contrôle la validité du certificat** (au sens des PKI) ;
- **recherche l'identité** de l'utilisateur, en s'appuyant sur un annuaire LDAP ou une base de données.

4.2.3 Sur un domaine Kerberos

Ce mode n'est pas implémenté dans la version courante de *GenericHandler*, mais l'authentification *Kerberos* fonctionne dans plusieurs universités américaines.

4.2.4 Sur un domaine NIS

Les organisations ne disposant pas (encore) d'un annuaire LDAP peuvent utiliser CAS en vérifiant l'authentification des utilisateurs sur un domaine NIS.
La connexion du serveur CAS au domaine NIS est fait soit directement en spécifiant un ou plusieurs serveurs NIS (redondants), soit par *broadcast* si aucun serveur n'est spécifié.

4.2.5 Sur domaine Windows NT

CAS peut également déléguer l'authentification à un domaine *Windows NT*. La connexion du serveur CAS au domaine *Windows NT* se fait vers un ou plusieurs contrôleurs de domaine (redondants, natifs NT ou Samba).

4.2.6 Sur un fichier

GenericHandler offre enfin (à l'origine à des fins de test, mais cette méthode d'authentification peut être utilisée en production pour des cas très particuliers) la possibilité d'authentifier les utilisateurs sur un fichier d'utilisateurs de type *passwd Unix* (générés par *htpasswd* par exemple).

---

[6] En cas d'authentification infructueuse sur un des annuaires, l'utilisateur n'est pas authentifié puisque les annuaires sont sensés contenir les même données.
[7] Les gestionnaires *MySQL* et *PostGreSQL* sont supportés et le support de *Oracle8* est en cours de développement.

5 CAS-ification d'une application *web*

La CAS-ification d'une application *web* est une chose aisée et légère. Des librairies clientes sont disponibles afin de faciliter cette opération.

On distingue trois types d'applications :

- **les applications client CAS « simples »** : elles ont juste besoin d'authentifier une personne ;
- **les applications mandataires (ou proxies) CAS** : elles on besoin d'authentifier la personne, et font également appel à des services tiers ayant besoin d'authentifier (un *web service*, par exemple). Ces applications doivent donc demander un *Proxy Granting Ticket* auprès du serveur CAS pour cet utilisateur ; ce PGT permettra à l'application de requérir ultérieurement un *Proxy Ticket* qu'elle passera au service tiers ;
- **les services tiers**, qui obtiennent un *Proxy Ticket* d'un mandataire CAS ; ce PT sera exploité par le service tiers de la même façon qu'un *Service Ticket*, afin d'obtenir l'identité de l'utilisateur.

5.1 Les applications client CAS « simples »

Le principe est de disposer d'une fonction ou méthode, qui, lorsqu'elle est appelée, déroule le mécanisme d'authentification CAS et retourne l'identifiant de l'utilisateur.

Cette fonction doit donc réaliser les opérations suivantes :

- si l'utilisateur n'est pas déjà authentifié, **rediriger le navigateur *web* vers le serveur CAS**, en passant en paramètre une URL de retour ;
- être en attente sur l'URL de retour, afin de **récupérer un *Service Ticket*** ;
- **valider le *Service Ticket* auprès du serveur CAS** en générant une requête HTTP(S). Le serveur CAS renvoie alors l'identifiant de l'utilisateur, certifié.

Afin d'illustrer la simplicité du code nécessaire, un exemple PHP réalisant ces opérations est donné ci-après. Dans une application réelle, c'est une librairie qui serait utilisée, comme décrit ensuite.

5.1.1 Un exemple simple en PHP

Nous montrons ci-dessous comment un client CAS (`script.php`) peut être simplement écrit en PHP.

Si ce script est appelé sans paramètre (par ex. `http://test.univ.fr/script.php`), il redirige le navigateur *web* vers le serveur CAS, en passant sa propre URL en paramètre ; l'URL d'appel est alors la suivante :

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

L'utilisateur s'authentifie auprès du serveur CAS ; le navigateur est redirigé vers l'URL de retour, avec en paramètre le *Service Ticket* . l'URL de retour est alors (par exemple) :

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

Le script `script.php` va alors valider ce ticket auprès du serveur CAS, directement en HTTP(S) ; l'URL appelée est :

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

Le serveur CAS valide ce ticket, et retourne alors l'identité de la personne, dans un format XML :

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

Un code possible pour l'exemple décrit ci-dessus est le suivant :

```
<?php  /* ------ exemple de client CAS écrit en PHP --------*/

  // localisation du serveur CAS
  define('CAS_BASE','https://auth.univ.fr');

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

```php
  /** Cette simple fonction réalise l'authentification CAS.
   * @return  le login de l'utilisateur authentifié, ou FALSE.
   */
  function authenticate() {
      global $service ;

      // récupération du ticket (retour du serveur CAS)
      if (!isset($_GET['ticket'])) {
          // pas de ticket : on redirige le navigateur web vers le serveur CAS
          header('Location: ' . CAS_BASE . '/login?service='  . $service));
          exit() ;
      }
      // un ticket a été transmis, on essaie de le valider auprès du serveur CAS
      $fpage = fopen (CAS_BASE . '/serviceValidate?service='
                              . preg_replace('/&/','%26',$service) . '&ticket=' . $ticket,  'r');
      if ($fpage) {
          while (!feof ($fpage)) { $page .= fgets ($fpage, 1024); }
          // analyse de la réponse du serveur CAS
          if (preg_match('|<cas:authenticationSuccess>.*</cas:authenticationSuccess>|mis',$page)) {
              if(preg_match('|<cas:user>(.*)</cas:user>|',$page,$match)){
                  return($match[1]);
              }
          }
      }
      // problème de validation
      return FALSE;
  }

  $login = authenticate();

  if ($login === FALSE ) {
      echo 'Requête non authentifiée (<a href="'.$service.'"><b>Recommencer</b></a>).';
      exit() ;
  }

  // à ce point, l'utilisateur est authentifié
  echo 'Utilisateur connecté : ' . $login . '(<a href="' . CAS_BASE . '/logout"><b>déconnexion</b></a>)';
?>
```

5.1.2 *Utilisation de la librairie* CASUtils

CASUtils est une librairie Perl, fournie avec la distribution cliente CAS. Voici un exemple de script Perl (`testcas.cgi`) utilisant cette librairie :

```perl
#!/usr/bin/perl

use lib "/var/www/cgi-bin";
use CASUtils;
require CGI;

my $cgi = new CGI;
my $status = &Yale::CASUtils::check_login('http://test/univ.fr/cgi-bin/testcas.cgi');
if ($status == -1) {
    printf "Content-type: text/plain\n\nRequête non authentifiée";
    exit 0;
}

printf "Content-type:text/plain\n\nUtilisateur connecté : $status";
```

La librairie *CASUtils* est rudimentaire (URLs relatives au serveur CAS codées « en dur » dans la librairie) mais permet de CAS-ifier facilement n'importe quelle application écrite en *Perl*.

5.1.3 *Utilisation de la librairie* phpCAS

La librairie *phpCAS* [14] a été développée par le projet *ESUP-Portail*. En voici un exemple d'utilisation :

```php
<?php  /* ------ exemple de client CAS utilisant phpCAS --------*/
   include_once('CAS.php');
   phpCAS::client(CAS_VERSION_2_0,'cas.univ.fr',443,'');
   phpCAS::authenticateIfNeeded();
?>
<html>
 <body>
   <h1>Authentification réussie !</h1>
   <p>L'utilisateur authentifié est <b><?php echo phpCAS::getUser(); ?></b>.</p>
 </body>
</html>
```

5.1.4 Autre librairies

D'autres librairies sont fournies avec la distribution client CAS, pour différents langages : ASP, *Java*, JSP, PL/SQL, et *Python*. Elles ne sont pas détaillées ici, mais leur usage est aisé.

5.2 Les applications mandataires (*proxies*) CAS

Le début du mécanisme est identique à celui d'un client CAS ordinaire : la récupération d'un *Service Ticket*.

Ensuite, lors de la validation de ce ST, un paramètre supplémentaire est passé au serveur CAS : une URL de *callback*. Le serveur CAS retourne alors :
- l'identifiant de l'utilisateur (comme pour un client CAS non mandataire),
- un PGT par l'URL de *callback*[8].

Comme vu en 3.3.2 (« *Fonctionnement 2-tiers* »), ce PGT permet ultérieurement au mandataire de s'authentifier auprès du serveur CAS pour l'utilisateur (identifié par le ST).

Les librairies clientes *Java* et *phpCAS* permettent de masquer la complexité lors du développement d'un mandataire CAS. Voici par exemple l'implémentation d'un mandataire CAS à l'aide de la librairie *phpCAS* :

```
<?php /* ------ exemple de mandataire CAS utilisant phpCAS --------*/
  include_once('CAS.php');
  phpCAS::proxy(CAS_VERSION_2_0,'auth.univ.fr',443,'');
  phpCAS::authenticateIfNeeded();
?>
<html><body>
<p>le login de l'utilisateur est <b><?php echo phpCAS::getUser(); ?></b>.</p>
<?php
  flush();
  // appel du service tiers
  if (phpCAS::serviceWeb('http://test.univ.fr/serviceWeb.php',$err_code,$output)) {
    echo 'Reponse du service tiers : <br>' . $output;
  }
?>
</body></html>
```

5.3 Les applications tierces

Les applications tierces sont aussi faciles à CAS-ifier qu'un client CAS simple. Elles font exactement le même travail qu'un client CAS simple, en validant auprès du serveur CAS *Proxy Ticket* (à la place d'un *Service Ticket* pour les clients CAS). Ainsi, le service tiers associé à l'exemple précédent serait, en phpCAS :

```
<?php  /* ------ exemple de service CAS utilisant phpCAS --------*/
  include_once('CAS.php');
  phpCAS::client(CAS_VERSION_2_0,'cas.univ.fr',443,'');
  phpCAS::authenticateIfNeeded();

  // pour cet exemple, on retourne une petite phrase qui indique que l'authentification est correcte
  echo '<p>le login utilisateur est <b>' . phpCAS::getUser() . '</b>.</p>';
?>
```

5.4 Quelques précautions à prendre lors de la CAS-ification d'applications

5.4.1 Gérer des sessions applicatives

Les applications doivent gérer des sessions applicatives, pour que le mécanisme CAS ne soit déroulé qu'une seule fois lors de l'accès à l'application, et non à chaque accès, ceci pour des raisons évidentes de performances. Cette remarque vaut également pour les échanges entre mandataires CAS et services tiers.

5.4.2 Risque d'asynchronisme de sessions (pour les mandataires CAS)

On a vu que la récupération d'un PGT pour un utilisateur, et de PT est une chose facile, en utilisant les libraires adéquates. Le développeur d'une application mandataire doit cependant prendre certaines précautions. Prenons l'exemple d'un portail *web*, qui serait mandataire CAS.

L'utilisateur se « connecte » dans son portail ; il s'authentifie donc via le serveur CAS, et le portail récupère un PGT propre à cet utilisateur. Une session applicative est mise en place entre le portail et l'utilisateur, session qui peut durer plusieurs heures. Imaginons que, pendant cette session applicative, le PGT devienne invalide (expiration du PGT, déconnexion CAS

[8] Le PGT est transmis par le serveur CAS grâce à l'URL de *callback*, un canal indépendant et chiffré, ce qui assure sa sécurité.

depuis une autre fenêtre de navigateur, ...) ; on se trouve alors dans un cas de figure où l'utilisateur a une session active auprès le portail, celui-ci étant dans l'impossibilité de récupérer un PT pour accéder à un service tiers.

Ce cas de figure doit être traité par les applications mandataires, par une déconnexion forcée de l'utilisateur.

5.5 Authentification CAS pour des pages web statiques

Nous avons vu que CAS apportait au développeur des librairies pour authentifier les utilisateurs d'applications *web*. CAS permet également de protéger des documents statiques, grâce au module Apache *mod_cas*. À l'aide de simples directives de configuration d'Apache, un site ou des parties d'un site peuvent n'être rendues accessibles qu'après authentification auprès d'un serveur CAS[9]. On utilisera par exemple les directives :

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

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

6 CAS-ification d'une application non *web*

L'objectif premier d'un mécanisme de SSO *web* est bien sûr ... d'offrir un service d'authentification unique simple et performant à des applications *web*. CAS va bien plus loin en permettant de rendre compatibles différents services non *web*[10] tels que IMAP, FTP, ... avec son propre mécanisme.

Il suffit pour cela que ces services s'appuient sur PAM (*Pluggable Authentication Module*) [10], comme savent le faire la plupart des services de base sous *Unix*.

6.1 Le module PAM *pam_cas*

Le module *pam_cas* est fourni avec la distribution client CAS. Il est à la fois très puissant et léger (environ 300 lignes de code C, dont les deux tiers partagés avec le module Apache *mod_cas*).

Il permet ainsi à un service *Unix* d'authentifier une personne en recevant un identifiant (comme d'habitude) et un ticket (*Service Ticket* ou *Proxy Ticket*) à la place du mot de passe. Le ticket reçu par le service est validé par *pam_cas* auprès du serveur CAS[11].

Notons que l'on ne conçoit l'utilisation de *pam_cas* que dans un fonctionnement multi-tiers, le service *Unix* n'étant accédé que par un client (mandataire) CAS, qui fournit au service un *Proxy Ticket*. Il est en effet inconcevable de demander à un utilisateur humain (d'un service FTP par exemple) un ticket CAS.

Heureusement, le fonctionnement modulaire de PAM permet d'ajouter à un service un mode d'authentification tout en conservant les autres modes, comme le montre l'exemple suivant.

6.2 Exemple : utilisation de pam_cas pour un serveur IMAP

L'objectif est de rendre un serveur IMAP compatible avec CAS, afin d'autoriser des accès depuis un portail *web*, ou un *webmail*, tout en continuant à autoriser les clients de messagerie traditionnels à utiliser le mot de passe utilisateur, et non un ticket.

Sous réserve que le serveur IMAP soit compatible avec PAM (ce qui est généralement le cas), le fichier de configuration de PAM pour IMAP pourra par exemple ressembler à :

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

[9] Les utilisateurs non authentifiés se voient redirigés vers le serveur CAS, comme lors de l'accès à une application. En ce sens, *mod_cas* est un client CAS.
[10] L'accès à ces services non web doit néanmoins se faire à partir d'une application *web* (un mandataire CAS).
[11] Afin de limiter les requêtes vers le serveur CAS, la validation par *pam_cas* n'est effectuée que lorsque le mot de passe fourni a la forme d'un ticket (commence par « ST- » ou « PT- »).

Dans notre exemple, une première authentification est tentée via *pam_cas* ; si cette authentification échoue, une seconde tentative est tentée via *pam_ldap* ; enfin, en cas de nouvel échec, une authentification *Unix* traditionnelle va être tentée.

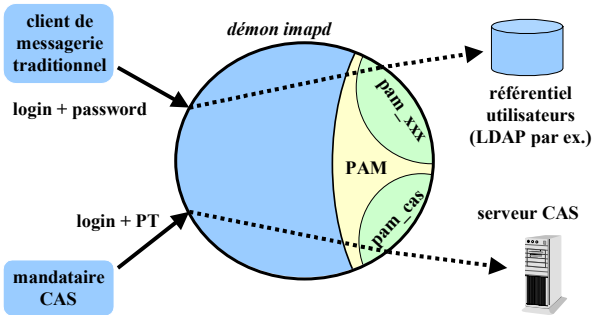

Figure 10 : *Utilisation de* pam_cas *pour un serveur IMAP*

6.3 CAS-ification du serveur Cyrus-IMAP

Le cas du protocole IMAP est très particulier, et certainement un des plus délicats à mettre en oeuvre : les clients IMAP et principalement les *webmails* ont la fâcheuse tendance à générer de très nombreuses requêtes, avec rupture et ré-ouverture de connexions, donc nouvelles demandes d'authentification.

Dans le cadre d'un simple *webmail*, cela a pour conséquence une charge plus importante du serveur *web*. Dans le cadre d'une authentification CAS, puisqu'à chaque requête IMAP est associée une authentification, la charge est également supportée par le serveur CAS, ce qui peut être rédhibitoire.

En l'absence d'un mécanisme de cache côté serveur IMAP, et pour chaque nouvelle connexion, l'application cliente (le *webmail*) doit demander un *Proxy Ticket* auprès du serveur CAS, puis le module *pam_cas* doit valider ce ticket. Le mécanisme est simple à mettre en oeuvre, mais on en voit rapidement la limite en terme de performances. Un cache est indispensable.

La mise en œuvre d'un tel cache est possible, en standard, avec le serveur *Cyrus*-IMAP (cf Figure 11). En effet, *Cyrus*-IMAP s'appuie sur *Cyrus*-SASL pour l'authentification ; or, *Cyrus*-SASL peut utiliser directement différents mécanismes d'authentification (PAM, LDAP, *Kerberos*, fichier DBM, ...) ou encore faire appel à un démon *Unix* dédié à cet effet, *saslauthd*.

Ce démon, qui communique avec *Cyrus*-SASL via une *socket Unix*, présente un double intérêt :
- **il peut être exécuté par un super-utilisateur** (*root*), ce qui permet l'authentification *Unix*, tout en continuant à exécuter le démon IMAP avec un compte à privilèges limités (pour des raisons évidentes de sécurité).
- **il propose un mécanisme de cache**.

Grâce au cache de *saslauthd*, l'application cliente IMAP est capable de rejouer le même *Proxy Ticket*, *saslauthd* n'ayant plus besoin de faire appel à PAM pour s'assurer de la validité du ticket une fois celui-ci stocké dans son cache (flèche 2 Figure 11). L'utilisation du cache permet de solliciter au minimum le serveur CAS[12].

[12] La mise en œuvre du cache de *saslauthd* a montré une efficacité de plus de 95% (moins de 5% des PT donnent lieu à une validation auprès du serveur CAS, soit une réduction de sa charge d'un facteur supérieur à 20).

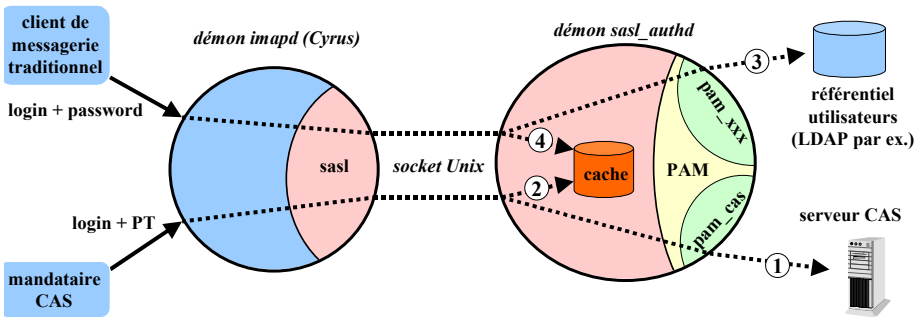

Figure 11 : *CAS-ification du serveur* Cyrus-*IMAP*

6.4 Application : *webmail* en SSO avec IMP

Le cas du *webmail* est très difficile à traiter du fait des nombreuses requêtes d'ouverture de connexion générées.

Dans un premier temps, il faut adapter le *webmail* (en l'occurrence, IMP) en lui donnant les fonctionnalités de mandataire CAS. C'est une chose aisée, grâce à la librairie *phpCAS* (cf 5.2 « *Les applications mandataires (proxies) CAS* »).
Il devient donc capable d'obtenir facilement un *Proxy Ticket* pour permettre au service IMAP d'authentifier l'utilisateur.

Il faut ensuite modifier le comportement du *webmail* pour tenir compte de la versatilité de ce nouveau type de mot de passe. En effet, le PT est manipulé comme un mot de passe par le *webmail*, mais sa durée de validité est limitée. Le *webmail* peut ainsi être amené à utiliser plusieurs fois le même PT (grâce au cache IMAP), comme un mot de passe traditionnel.

Malheureusement, il n'est pas exclu qu'un utilisateur, durant sa session *webmail*, lance également une requête IMAP par une autre voie, son portail d'accès ou son client de messagerie habituel par exemple. Dans ce cas, le PT qui était mémorisé par le cache IMAP est écrasé par un autre PT, voire par le mot de passe de l'utilisateur.

Lors de la tentative suivante de la part du *webmail* avec l'ancien PT, la connexion IMAP est refusée. Il faut alors que le *webmail* soit capable de traiter ce refus sans clore la session de l'utilisateur, c'est-à-dire de redemander un nouveau PT et de retenter une nouvelle connexion IMAP.

Ce problème, complexe à comprendre, n'en est pas moins difficile à résoudre ;-). Les librairies clientes sont de ce fait beaucoup plus complexes que nous l'avons laissé penser en 5.1.1 (« *Un exemple simple en PHP* »).

7 Limitations actuelles et perspectives

Nous avons vu tout l'intérêt de CAS pour mettre en œuvre une plate-forme de SSO :
- open-source et libre,
- très bon niveau de sécurité,
- mise en oeuvre aisée du serveur CAS,
- fonctionnement multi-tiers,
- CAS-ification aisée des applications web.

Nous abordons maintenant les limitations de CAS, ou certains points plus délicats, et les perspectives permettant un certain optimisme.

7.1 CAS est un SSO, limité à une authentification locale

CAS est proposé comme mécanisme de SSO web, et nous avons vu qu'il sait aller au delà, grâce à *pam_cas*. En revanche, il se limite à l'authentification des utilisateurs ; Il ne traite absolument pas les besoins liés aux autorisations (droits applicatifs) ni au transport d'attributs[13].

En outre, la base d'authentification est locale, au niveau de l'établissement. Les aspects inter-établissements ne sont donc pas pris en compte.

[13] Dans les recommandations du groupe de travail AAS [16] (annexe du SDET [17]), la gestion des autorisations est confié à un service dédié, comme l'est le serveur CAS pour l'authentification.

Le projet *Shibboleth* [19] est un projet open-source d'*Internet2* très actif qui propose un mécanisme de transport d'attributs et d'authentification inter-établissements. Il ne propose pas pour le moment le SSO local, laissé à l'initiative de chaque établissement. De récentes discussions autour de *Shibboleth* laissent entendre qu'un mécanisme de SSO pourrait être proposé par défaut ; CAS fait partie des implémentations pressenties pour cela, ce qui pallierait les limitations actuelles[14].

7.2 Montée en charge et tolérance aux pannes

Dans un environnement CAS, tout accès à une application web nécessitant authentification passe nécessairement par le serveur CAS. Sa disponibilité et son bon fonctionnement deviennent critiques pour le fonctionnement des applicatifs.

Dans l'état actuel, CAS ne permet pas d'envisager un partage de charge entre plusieurs serveurs. En effet, les différents objets manipulés (essentiellement les tickets) sont maintenus en mémoire du processus serveur CAS, ceci a priori pour des raisons d'efficacité et de simplicité. Ce fonctionnement rend impossible le partage de charge (*load balacing*) entre plusieurs serveurs rendant le service d'authentification[15].

Dans la pratique, les universités ayant déployé CAS n'ont pas rencontré de problème de montée en charge, les processus mis en œuvre étant relativement simples. Le problème de la tolérance aux fautes est en revanche, au vu de la criticité du service d'authentification, beaucoup plus crucial.

Il est certes possible de maintenir en état de veille un serveur CAS « de secours » (*spare*), qui pourra être utilisé en cas de défaillance du serveur principal ou simplement de maintenance. Dans le cas, par exemple, où un serveur Apache est mis en frontal du serveur J2EE (*Tomcat* au autre) supportant le serveur CAS ; la bascule d'un serveur CAS vers un autre est extrêmement simple.
Elle n'est cependant pas transparente : tous les tickets valides (TGC et PGT notamment) sont alors perdus, ce qui entraîne une ré-authentification de l'utilisateur lors de l'accès à une nouvelle application ou un nouveau service.

Une solution, consistant à mémoriser les tickets de type *granting* (TGC et PGT) dans une base de donnée, serait facilement envisageable[16]. Dans ce cas, la bascule d'un serveur CAS à un autre (s'appuyant sur la même base de données) aurait des conséquences négligeables.
Ceci permettrait d'envisager plus sereinement une maintenance sur la machine hébergeant le serveur CAS.

Références

[1] Autorité de certification CNRS. http://igc.services.cnrs.fr/CNRS-Test/
[2] Autorité de certification du CRU, http://pki.cru.fr
[3] Introduction aux architectures web de Signe Sign-On, Olivier Salaun, JRES2003
[4] Persistent client state (HTTP cookies). http://wp.netscape.com/newsref/std/cookie_spec.html
[5] Sun One Identity Server. http://www.sun.com
[6] Microsoft .NET Passport: One easy way to sign in online. http://www.passport.com
[7] ITS Central Authentication Service, http://www.yale.edu/tp/cas/
[8] Kerberos: The Network Authentication Protocol, http://web.mit.edu/kerberos/www/
[9] JASIG (Java Architectures Special Interest Group), Evolving portal implementations. http://mis105.mis.udel.edu/ja-sig/uportal/
[10] Linux-PAM: Pluggable Authentication Modules for Linux, www.us.kernel.org/pub/linux/libs/pam/Linux-PAM-html/
[11] Web Services, http://www.w3.org/2002/ws/
[12] ESUP-Portail, http://www.esup-portail.org
[13] CAS GenericHandler, http://esup-casgeneric.sourceforge.net
[14] PhpCAS, http://esup-phpcas.sourceforge.net
[15] The Horde Project, http://www.horde.org
[16] Recommandations pour la gestion de l'authentification-autorisation-SSO, Juillet 2003, http://www.educnet.education.fr/equip/sdet.htm
[17] Schéma Directeur des Espaces numériques de Travail, juillet 2003, http://www.educnet.education.fr/equip/sdet.htm
[18] Sympa – Mailing List Manager, http://www.sympa.org
[19] The Shibboleth Project, http://shibboleth.internet2.edu/

[14] Les travaux récents de Serge Aumont sur Sympa [18] montrent qu'il est possible pour un client CAS de s'appuyer sur plusieurs serveurs CAS, mais sans aucune communication ni relation de confiance entre ces serveurs.
[15] Les tickets étant stockés en mémoire, il faut qu'un ticket soit validé par le serveur qui l'a émis.
[16] Cette solution serait en effet très peu pénalisante en terme de performances, car les tickets de type *granting* sont beaucoup plus rares que les PT et ST.