Retour de l'URN sur mise en place de CAS 6.6.9

Contexte

Eté 2023, l'Université de Rouen Normandie a procédé à une montée de version de son service d'authentification CAS en **6.6.9** (puis très rapidement en 6.6.10)

Cette page wiki vient en complément de nos pages **Retour de l'URN sur mise en place de CAS 6.0.4** et Retour de l'URN sur mise en place de CAS 6.4.1 ; les retours que l'on avait faits sur ces versions 6.x sont a priori toujours d'actualité.

Cette installation est faite sur une debian en utilisant **CAS Initializr** ; jusque-là notre installation et montée de version s'appuyait sur ce qui était utilisé /embarqué par le Kit installation CAS V5.2 AMU.

On a également profité de cette montée de version pour mettre en place le module OpenID Connect Authentication permettant à CAS de supporter l'authentification OpenIdConnect (oidc) en tant que fournisseur (provider/serveur).

En fin de document, on fait aussi un retour sur un problème rencontré avec le ticket registry sous Redis et cette version de CAS 6.6.9 ; on en profite pour documenter notre passage au ticket registry sous MongoDB.

Installation

Paquets

```
apt install openjdk-11-jdk-headless tomcat9 curl unzip
```

CAS Initializr

Pour construire notre CAS, on ajoute dans le .bashrc d'un utilisateur 'cas' :

```
function getcas() {
  url="https://casinit.herokuapp.com/starter.tgz"
  projectType="cas-overlay"
  dependencies=""
  directory="overlay"
  for arg in $@; do
    case "$arg" in
    --url|-u)
      url=$2
      shift 1
      ;;
    --type|-t)
      projectType=$2
      shift 1
      ;;
    --directory|--dir|-d)
      directory=$2
      shift 1
      ;;
    --casVersion|--cas)
      casVersion="-d casVersion=$2"
      shift 1
      ;;
    --bootVersion|--springBootVersion|--boot)
      bootVersion="-d bootVersion=$2"
      shift 1
      ;;
    --modules|--dependencies|--extensions|-m)
      dependencies="-d dependencies=$2"
      shift 1
      ;;
    *)
      shift
      ;;
    esac
  done
  rm -Rf ./${directory}
  echo -e "Generating project ${projectType} with dependencies ${dependencies}..."
  cmd="curl ${url} -d type=${projectType} -d baseDir=${directory} ${dependencies} ${casVersion} ${bootVersion}
| tar -xzvf -"
  echo -e "${cmd}"
  eval "${cmd}"
  ls
}
```

Puis on créé un fichier urn-generate-cas-overlay.sh :

```
#!/bin/bash -x
getcas --cas 6.6.9 --modules support-ldap,support-json-service-registry,support-redis-ticket-registry,support-
spnego-webflow,support-trusted-mfa-mongo,support-throttle,support-reports,supp
ort-interrupt-webflow,support-oidc
```

On lance ce script qui nous construit l'overlay CAS dans le répertoire overlay. Ce répertoire est versionné via git avec pour premier commit "Project created by Apereo CAS Initializr".

On ajuste le build.gradle pour installer notamment les modules CAS d'esup-otp et esup-agimus ; on fixe aussi le look via l'ajout de css et surcharge de templates.

```
cas@cas:/opt/overlay$ git diff
warning: LF will be replaced by CRLF in build.gradle.
The file will have its original line endings in your working directory
warning: LF will be replaced by CRLF in gradle.properties.
The file will have its original line endings in your working directory
diff --git a/build.gradle b/build.gradle
index bb8758e..6cffd74 100644
--- a/build.gradle
+++ b/build.gradle
```

```
@@ -82,6 +82,9 @@ repositories {
         }
         mavenContent { releasesOnly() }
     }
+    maven {
+        url "https://jitpack.io"
+    }
 }


@@ -273,11 +276,15 @@ dependencies {
     implementation "org.apereo.cas:cas-server-support-json-service-registry"
     implementation "org.apereo.cas:cas-server-support-redis-ticket-registry"
     implementation "org.apereo.cas:cas-server-support-spnego-webflow"
+    implementation files("${projectDir}/lib/jcifs-ext.jar")
     implementation "org.apereo.cas:cas-server-support-trusted-mfa-mongo"
     implementation "org.apereo.cas:cas-server-support-throttle"
     implementation "org.apereo.cas:cas-server-support-reports"
     implementation "org.apereo.cas:cas-server-support-interrupt-webflow"
     implementation "org.apereo.cas:cas-server-support-oidc"
+    implementation "com.github.vbonamy:cas-server-support-agimus-cookie:cas-6.4.x-SNAPSHOT"
+    implementation "com.github.vbonamy:cas-server-support-agimus-logs:cas-6.6.x-SNAPSHOT"
+    implementation "com.github.EsupPortail:esup-otp-cas:6.6.x-SNAPSHOT"


     if (project.hasProperty("casModules")) {
diff --git a/gradle.properties b/gradle.properties
index e33296a..d875561 100644
--- a/gradle.properties
+++ b/gradle.properties
@@ -29,7 +29,7 @@ dockerImagePlatform=amd64:linux
 # Include launch script for executable WAR artifact
 # Setting this to true allows the final web application
 # to be fully executable on its own
-executable=true
+executable=false



@@ -37,7 +37,7 @@ executable=true
 # if the overlay application supports or provides the chosen type.
 # You should set this to blank if you want to deploy to an external container.
 # and want to set up, download and manage the container (i.e. Apache Tomcat) yourself.
-appServer=-tomcat
+appServer=

 # Settings to generate keystore
 # used by the build to assist with creating


cas@cas:/opt/overlay$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   build.gradle
        modified:   gradle.properties

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        lib/
        src/main/resources/static/
        src/main/resources/templates/

no changes added to commit (use "git add" and/or "git commit -a")



cas@cas:/opt/overlay$ tree src/main/resources/static/ src/main/resources/templates/ lib/
src/main/resources/static/
??? images
```

```
    ??? cas-logo.png
    ??? logo_universite_rouen.svg
src/main/resources/templates/
??? fragments
?   ??? footer.html
?   ??? pmlinks.html
??? layout.html
lib/
??? jcifs-ext.jar

2 directories, 6 files
```

Puis on commit.

Packaging

Depuis /opt/overlay on lance le packaging via gradle :

```
./gradlew clean build
```

Configuration du tomcat

On a choisi ici l'utilisation du tomcat9 fourni par debian, on modifie /var/lib/tomcat9/conf/server.xml pour ne pas loguer les accès (qui seront logués par le apache en frontal), désactiver le connecteur http et activer le connecteur ajp.

Déploiement

```
rm -rf  /var/lib/tomcat9/webapps/ROOT && unzip /opt/overlay/build/libs/cas.war  -d /var/lib/tomcat9/webapps/ROOT
```

Configuration du démarrage et démarrage

```
systemctl enable tomcat9

systemctl restart tomcat9
```

Installation des autres composants

On installe et configure également les autres paquets/briques nécessaires au bon fonctionnement de notre CAS, à savoir :

- apache avec ajp, ssl, ...
- certbot pour automatiser le renouvellement des certificats
- les briques requises par les modules CAS utilisés : mogodb, redis, etc.
- les briques esup-otp et dépendances associées que l'on installe sur le même serveur

Ajustement des configurations CAS 6.4.6.3 6.6.9

À chaque mise à jour de CAS, son lot de renommage de configurations ... liste non exhaustive pour CAS dans /etc/cas/config/cas.properties :

- cas.audit.* cas.audit.engine.*
- suppression de cas.authn.attributeRepository.ldap[0].useSsl et cas.authn.attributeRepository.ldap[0].providerClass
- cas.authn.attributerepository.maximum-cache-size cas.authn.attributerepository.core.maximum-cache-size
- cas.authn.mfa.trusted.authentication-context-attribute cas.authn.mfa.trusted.core.authentication-context-attribute
- cas.authn.mfa.trusted.device-registration-enabled cas.authn.mfa.trusted.core.device-registration-enabled
- suppression de cas.authn.mfa.trusted.expiration=7 et cas.authn.mfa.trusted.timeUnit=DAY

Concernant ce dernier point, les paramétrages d'activation et expiration de l'enregistrement des navigateurs/périphériques clients (devices) comme étant sûres (trusted) sont portés par l'implémentation de la MFA.
Dans le cadre d'esup-otp, dans le fichier esupotp.properties on peut ainsi positionner par exemple :

```
esupotp.trustedDeviceEnabled=true
esupotp.isDeviceRegistrationRequired=false
esupotp.deviceRegistrationExpirationInDays=7
```

Les fichiers des déclarations de services dans /etc/cas/services doivent aussi être modifiés :

```
find /etc/cas/services/ -name "*json" -exec sed -i 's/org.apereo.cas.services.RegexRegisteredService/org.apereo.
cas.services.CasRegisteredService/' {} \;
```

Pour esup-otp, dans /etc/cas/config/esupotp.properties, suppression de esupotp.byPassServicesIfNoEsupOtpMethodIsActive

OpenIdConnect

Configuration OpenIdConnect sur CAS

Ajout de la configuration suivante dans /etc/cas/config/cas.properties :

```
##############################
## OPENID CONNECT           ##
##############################
cas.authn.oidc.core.issuer=https://cas.univ-rouen.fr/oidc
```

Attention ici à vérifier que ce paramétrage est cohérent avec le paramétrage du nom du serveur CAS (cf ci-dessous, pas de :443 en plus, pas de chemin relatif, ... à adapter si vous avez /cas en plus) ; une incohérence peut poser problème pour la mise en place de oidc ; CAS doit a priori vérifier de manière basique que les propriétés concordent bien (simple equals sur des chaines de caractères ?) :

```
cas.server.name: https://cas.univ-rouen.fr
cas.server.prefix: https://cas.univ-rouen.fr
```

Pour que cela fonctionne, CAS doit pouvoir écrire le fichier /etc/cas/config/keystore.jwks au travers de tomcat lors du lancement.

Sous debian, il faut modifier /etc/systemd/system/multi-user.target.wants/tomcat9.service pour y insérer le paramétrage suivant :

```
ReadWritePaths=/etc/cas/config/keystore.jwks
```

On créé le fichier /etc/cas/config/keystore.jwks et on lui donne comme propriétaire l'utilisateur tomcat.

Lors du lancement, on prend en compte les messages du type "The generated key MUST be added to CAS settings" pour compléter la configuration openid.

Un service pourra alors être déclaré de cette façon :

```
cat /etc/cas/services/test_oidc-1000.json
{
  "@class" : "org.apereo.cas.services.OidcRegisteredService",
  "clientId": "clientoidctestaconfigurercoteclient",
  "clientSecret": "monsecretaconfigurercoteclient,
  "serviceId" : "https://test-oidc.univ-rouen.fr/secureoidc/redirect_uri",
  "name": "OIDC",
  "id": 1000,
}
```

Client OpenIdConnect avec Apache2 et mod_auth_openidc

Comment conseillé par les collègues de l'AMU (cf leur présentation à Esup-Days 26) nous avons validé le bon fonctionnement de CAS en tant que provider openidconnect via l'utilisation d'un apache avec mod_auth_openidc.

Ainsi le serveur derrière https://test-oidc.univ-rouen.fr pourra avoir pour configuration Apache :

```
ScriptAlias /secureoidc /var/www/printenv.pl
OIDCProviderMetadataURL https://cas.univ-rouen.fr/oidc/.well-known/openid-configuration
OIDCClientID clientoidctestaconfigurercoteclient
OIDCClientSecret monsecretaconfigurercoteclient
OIDCRemoteUserClaim sub
OIDCScope "openid email profile"
OIDCRedirectURI https://test-oidc.univ-rouen.fr/secureoidc/redirect_uri
#LogLevel info auth_openidc:debug
OIDCCryptoPassphrase ppassQuelconquePourChiffrementInterne
<Location /secureoidc>
 AuthType openid-connect
 Require valid-user
</Location>
```

/var/www/printenv.pl étant le "traditionnel" script utilisé notamment dans les configurations shibboleth et donnant simplement les entêtes HTTP.

```
#!/usr/bin/perl

print "Content-type: text/plain\n\n";

print "Variables d'environnement :\n\n";

foreach my $key (keys %ENV) {
        printf "$key=$ENV{$key}\n";
}
```

On y voit alors le REMOTE_USER de positionné, mais aussi des OIDC_CLAIM_* dont OIDC_CLAIM_attributes regroupant l'ensemble des attributs renvoyés (sous forme de json).

Ajustement script groovy déclenchement MFA - récupération de l'IP cliente

Lors de la migration sur ce nouveau CAS, il a fallu ajuster notre script groovy déclenchant la MFA (script donné derrière la configuration cas.authn.mfa.groovy-script.location).

Notre script tient compte notamment de l'IP du client pour déclencher ou non la MFA.

On récupérait jusque là cette IP par httpRequest.getRemoteAddr() - cependant, et depuis notre mise à jour CAS, ce script est maintenant aussi appelé lors de la validation du ticket par le service CAS. L'IP est alors l'ip du service et non pas du client/usager.

En ne demandant pas la MFA pour un client donné (au premier appel) mais en la demandant lors de la validation du ticket, nous récupérions une erreur de type **invalid_authentication_context** côté du service lors de la validation du ticket.

Pour récupérer l'IP du client/usager, il faut donc utiliser dans le script groovy.

```
def ip = authentication.attributes["clientIpAddress"].get(0)
```

et non pas (plus, en tout cas pour nous)

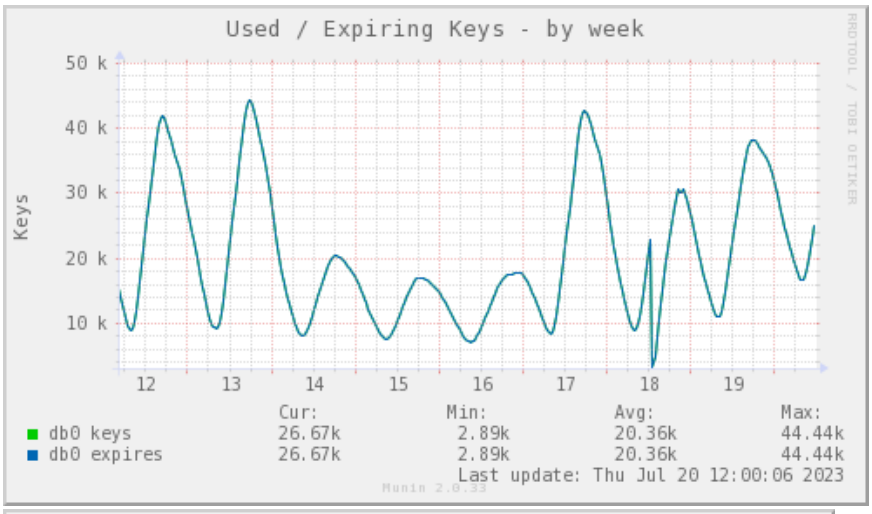
```
def ip = httpRequest.getRemoteAddr()
```

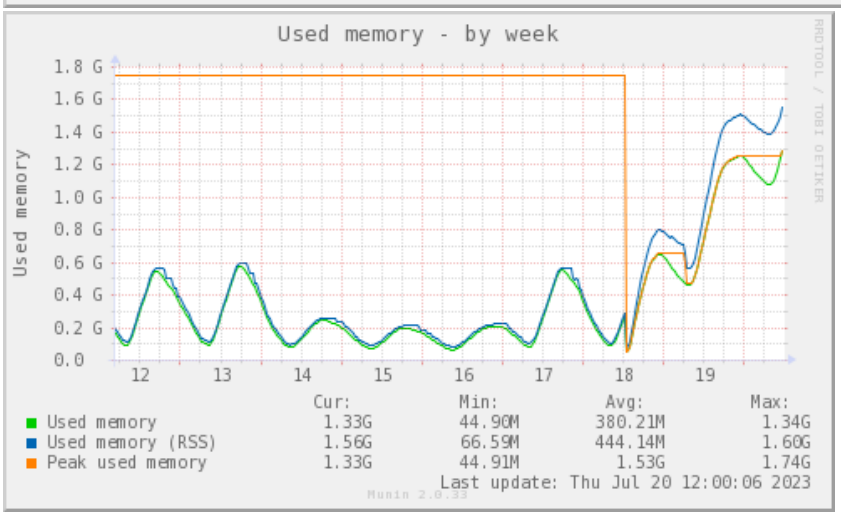
Cette revalidation, du fait que le client doit ou non passer par la (une) MFA (ré-exécution du script groovy) lors de la validation du ticket, est apparue entre la 6.4.6.6 et la 6.6.9

Ticket Registry

Ticket Registry Redis à éviter en CAS 6.6.9

2 jours après la migration sur CAS 6.6.9, on constate une augmentation de consommation mémoire dans le Redis.
La consommation mémoire va en s'accroissant ; la bascule sur la 6.6.9 a été faite le 18 juillet.





Dans Redis, en plus des traditionnels CAS_TICKET:TGT-numero persistants, on trouve maintenant autant de CAS_PRINCIPAL:uid que d'utilisateurs. Ces CAS_PRINCIPAL:uid sont des "sets" qui regroupent l'ensemble des TGT des utilisateurs.

On note que le TTL de ces CAS_PRINCIPAL:uid est de 14 jours (on propose le remember-me de 14 jours) et qu'il n'est de fait d'aucune utilité puisqu'il sera allongé à chaque authentification/connexion de l'usager (sur un espace de 14 jours, on peut estimer que la plupart des utilisateurs se connectent au moins une fois).

On peut espérer que le DefaultTicketRegistryCleaner appelle le ticketRegistry Redis pour supprimer les TGTs stockés dans ces clefs CAS_PRINCIPAL:uid ... mais force est de constater que ça ne semble pas être le cas sur notre installation.

Comme on avait pu le constater avec le JPA Ticket Registry (cf notre retour de 2019 à ce propos) le fonctionnement du Ticket Registry semble avoir été pensé pour déléguer les tâches d'expiration des données au système de stockage (registry) choisi.

Jusque-là le Redis Ticket Registry fonctionnait bien ainsi, mais l'apparition de ces entrées CAS_PRINCIPAL:uid (sorte de rustine ayant pour but de retrouver rapidement les tickets d'un utilisateur donné) vient casser ce système et il semble aventureux de fonctionner ainsi.

> Après lecture du code (DefaultTicketRegistryCleaner et RedisTicketRegistry dans CAS), ce que l'on en conclue ici, c'est que Redis supprime naturellement les entrées du type CAS_TICKET:TGT-numero en fonction du TTL positionné ; aussi lorsque le DefaultTicketRegistryCleaner passe, les tickets déjà supprimés par Redis lui-même ne sont pas listés/pris en compte par la procédure de CAS qui, de fait, ne les supprime pas des sets listés dans CAS_PRINCIPAL:uid ; en fonctionnant ainsi, l'ensemble des serialisation des TGT (avec l'ensemble des attributs utilisateurs associés) vont donc perdurer dans les CAS_PRINCIPAL:uid sans jamais être nettoyés pour consommer très rapidement l'ensemble de la RAM disponible (le graphe nous montre une consommation d'environ 1GB de RAM jour sur une période estivale de congés) - on peut considérer ce pb comme un bug.
> Provenant de toute manière d'une implémentation contournant une "limitation" Redis sur la recherche par index, il semble ici plus simple de ne pas s'obstiner sur cette option de ticket registry Redis, dans cette version 6.6 en tout cas.

Notre CAS avec esup-otp utilisant dès à présent pour certaines données une base MongoDB, on s'est naturellement tourné vers le Ticket Registry sous MongoDB.

Passage au Ticket Registry MongoDB

Dans le build.gradle de l'overlay on remplace

```
org.apereo.cas:cas-server-support-redis-ticket-registry
```

par

```
org.apereo.cas:cas-server-support-mongo-ticket-registry
```

Puis dans /etc/cas/config/cas.properties on commente les paramètres du ticket-registry redis pour ajouter ceux de mongodb

```
cas.ticket.registry.mongo.client-uri=mongodb://localhost/cas-ticketregistry
cas.ticket.registry.mongo.crypto.enabled=true
cas.ticket.registry.mongo.crypto.signing.key=clef-generee-au-premier-demarrage
cas.ticket.registry.mongo.crypto.encryption.key=clef-generee-au-premier-demarrage
```

Un redémarrage CAS suffit à passer sur le nouveau ticket registry, évidemment toutes les sessions sont perdues lors de cette bascule.

MongoDB (et surtout l'implémentation du TicketRegistry avec celui-ci) semble bien convenir :

- usage de TTL (expireAt) gérés et propres à MongoDB.
- possibilité de requêter facilement les tickets d'un utilisateur donné, et ce sans rustine (contrairement au ticket registry redis donc)
- par défaut sous debian, le mongodb a un stockage persistant (file storage sous /var/lib/mongodb/)

A l'usage, nous ne ressentons pas d'impact de performances par rapport au fonctionnement avec Redis.

On note par contre que la demande de suppression des sessions CAS d'un utilisateur (cf le script en place et partagé sur ce même espace wiki) est maintenant très rapide (était très lent précédemment).

On note aussi une consommation continue d'1 cpu pendant 30 secondes, ce à un intervalle régulier de 2 minutes ; cela est du au DefaultTicketRegistryCleaner, les paragraphes qui suivent traitent de celui-ci.

Nettoyage de tickets via DefaultTicketRegistryCleaner

CAS positionne donc bien un expireAt (index TTL, lié à un expireAfterSeconds) propre à MongoDB dans les enregistrements, notamment les TGT (collection ticketGrantingTicketsCollection).

Notre configuration autour des durées de vies des tickets est la suivante :

```
# 15 jours au max -> mongodb supprimera lui-même les tickets au bout de 15 jours si
nécessaire
cas.ticket.tgt.primary.max-time-to-live-in-seconds=1296000
# DefaultTicketRegistryCleaner supprimera les tickets non remember-me au bout de 8
heures
cas.ticket.tgt.primary.time-to-kill-in-seconds=28800

                                                    cas.ticket.tgt.remember-me.enabled=true ### Valeur
expiration tickets service/proxy : 10sec par défaut, on augmente car latence sur les sogo
...
cas.ticket.pt.timeToKillInSeconds=60
cas.ticket.st.timeToKillInSeconds=60

#
rememberme

cas.ticket.tgt.rememberMe.enabled=true
# DefaultTicketRegistryCleaner supprimera les tickets remember-me au bout de 14
jours
cas.ticket.tgt.remember-me.time-to-kill-in-seconds=1209600
# rememberme 2 semaines ok pour le
cookie

tgc.remember.me.maxAge=1209600
cas.tgc.rememberMeMaxAge=1209600
```

Les expireAt des TGT sont alors positionnés à la date du jour + 8H (cas.ticket.tgt.primary.time-to-kill-in-seconds) quand le RememberMe n'est pas actionné, et à la date du jour + 14 jours (cas.ticket.tgt.remember-me.time-to-kill-in-seconds) avec le RememberMe de positionné.
Les expireAfterSeconds (TTL) sont positionnés à 15 jours (cas.ticket.tgt.primary.max-time-to-live-in-seconds).

Aussi, les tickets sont supprimés par CAS lui-même via DefaultTicketRegistryCleaner en fonction de ce expireAt.
Et "au pire", ils sont supprimés par le mécanisme interne de MongoDB via le expireAfterSeconds 15 jours après le expireAt ... soit 15 jours et 8 heures après pour les TGT non remember-me et 29 (14+15) jours après pour les remember-me.
Cf la documentation MongoDB à ce sujet vis-à-vis de expireAfterSeconds.

Coût du nettoyage de tickets via DefaultTicketRegistryCleaner

Aussi, avec cette configuration ("par défaut"), DefaultTicketRegistryCleaner a la charge de supprimer tous les tickets expirés de la base. En soit, on peut penser que ce type d'opérations n'est pas gourmande : on supprime tous les tickets dont le expireAt est plus vieux que la date actuelle ; on peut penser que cela se fait en une seule requête. Malheureusement l'implémentation proposée ici est réalisée dans une logique de NoSQL générique : DefaultTicketRegistryCleaner récupère tous les tickets, les "lit" (et donc les décode si ils sont encodés, ce qui est le cas normalement si vous suivez les recommandations de CAS), sélectionne les tickets qu'il estime expirés pour enfin les supprimer. Cette implémentation générique est très coûteuse, d'autant que par défaut cette procédure de suppression de tickets par le DefaultTicketRegistry est appelée toutes les 2 minutes (cas.ticket.registry.cleaner.schedule.repeat-interval=PT2M). Ainsi, lors d'une période creuse sur notre CAS, on note une consommation d'environ **30 secondes de CPU** à chaque nettoyage de tickets (**toutes les 2 minutes**) résultant du déchiffrement de l'ensemble des tickets.

Une "optimisation" élémentaire est simplement d'augmenter le cas.ticket.registry.cleaner.schedule.repeat-interval à 1 heure par exemple (PT1H).

A l'autre extrême on peut aussi imaginer laisser le soin à MongoDB de s'occuper lui-même de supprimer les tickets expirés, d'autant que c'est tout l'intérêt de ce type de base et de la fonctionnalité d'index avec un expireAfterSeconds (ttl) de positionné et que c'est ce qui pose problème à l'implémentation via Redis avec l'introduction de l'entrée CAS_PRINCIPAL:uid correspondant à plusieurs TGT et ne pouvant de fait pas avoir un TTL propre.

De plus, la documentation CAS elle-même indique à propos du Ticket Registry Cleaner :

> The ticket registry cleaner is generally useful in scenarios where the registry implementation is unable to auto-evict expired tokens and entries on its own via a background task.

Tentative d'optimisation du nettoyage de tickets via MongoDB

Cf le parapgraphe suivant, on a préféré finalement ne pas procéder à ces paramétrages.

Pour laisser MongoDB s'occuper du nettoyage des tickets, on désactive simplement le nettoyage périodique des tickets par CAS ainsi :

```
cas.ticket.registry.cleaner.schedule.enabled=false
```

Cf ci-dessus cependant, le comportement ne sera cependant pas tout à fait adapté à ce qu'on recherche car CAS a positionné les expireAfterSeconds des index expireAt non pas à 0 seconde mais à la valeur donnée par cas.ticket.tgt.primary.max-time-to-live-in-seconds (15 jours).

Mettre cas.ticket.tgt.primary.max-time-to-live-in-seconds à 0 est tentant mais à 0 il n'est pas pris en compte. Le mettre à 1 seconde peut être aussi une idée, mais l'authentification dysfonctionne alors et il est rappelé dans les logs que cas.ticket.tgt.primary.max-time-to-live-in-seconds doit être plus grand que cas.ticket.tgt.primary.time-to-kill-in-seconds.

Nous n'avons pas trouvé la parade permettant via les configurations de mettre simplement ce expireAfterSeconds à 0.

Aussi, on les surcharche ainsi depuis mongosh :

```
db.proxyGrantingTicketsCollection.dropIndex('expireAt_1')
db.proxyTicketsCollection.dropIndex('expireAt_1')
db.serviceTicketsCollection.dropIndex('expireAt_1')
db.ticketGrantingTicketsCollection.dropIndex('expireAt_1')

db.proxyGrantingTicketsCollection.createIndex({ expireAt: 1 }, { expireAfterSeconds: 0 } )
db.proxyTicketsCollection.createIndex({ expireAt: 1 }, { expireAfterSeconds: 0 })
db.serviceTicketsCollection.createIndex({ expireAt: 1 }, { expireAfterSeconds: 0 })
db.ticketGrantingTicketsCollection.createIndex({ expireAt: 1 }, { expireAfterSeconds: 0 })
```

Les index ne seront pas écrasés par CAS ici lors du redémarrage car cas.ticket.registry.mongo.drop-indexes est à false et malgrè le cas.ticket.registry.mongo.update-indexes à true par défaut (on le met tout de même à false au passage), celui-ci n'a aucun effet si l'index a le même nom (expireAt ici).

Cependant, une fois ces modifications effectuées, on s'aperçoit que le expireAt (des tickets non remember-me) est maintenant fixé non plus à la valeur de cas.ticket.tgt.primary.time-to-kill-in-seconds mais à celle de cas.ticket.tgt.primary.max-time-to-live-in-seconds.
A ce jour, nous n'avons pas trouvé le code responsable de ce changement de comportement - on imagine que c'est lié à la désactivation du DefaultTicketRegistryCleaner.
On modifie le paramètre cas.ticket.tgt.primary.time-to-kill-in-seconds à 8H.

On a alors les paramètres suivants :

```
### Valeur d'expiration des
TGTs

# Set to a negative value to never expire
tickets

# 8 heures au max -> mongodb supprimera lui-même les tickets au bout de 8 heures (sauf remember-
me)
cas.ticket.tgt.primary.max-time-to-live-in-seconds=28800
# DefaultTicketRegistryCleaner supprimera les tickets non remember-me au bout de 8 heures (non utile -
désactivé)
cas.ticket.tgt.primary.time-to-kill-in-seconds=28800

### Valeur expiration tickets service/proxy : 10sec par défaut, on augmente car latence sur les sogo
...
cas.ticket.pt.timeToKillInSeconds=60
cas.ticket.st.timeToKillInSeconds=60

#
rememberme

cas.ticket.tgt.rememberMe.enabled=true
# mongodb (et pas DefaultTicketRegistryCleaner via notre config) supprimera les tickets remember-me au bout de
14 jours
cas.ticket.tgt.remember-me.time-to-kill-in-seconds=1209600
# rememberme 2 semaines ok pour le
cookie

tgc.remember.me.maxAge=1209600
cas.tgc.rememberMeMaxAge=1209600
# pinToSession = false pour que ça fonctionne même si chgt d'IP par
exemple.

cas.tgc.pinToSession=false

### Pas de Ticket Registry Cleaner -> mongo s'en
charge

cas.ticket.registry.cleaner.schedule.enabled=false
# index expireAt/expireAfterSeconds et plein texte adapté pour optimisation cpu
cas.ticket.registry.mongo.update-indexes=false
```

Effets de bord d'une telle modification

En simplifiant la gestion des expirations des tickets pour faire cette optimisation, on a cependant provoqué au moins un effet de bord dans notre installation qui amène une régression pour les utilisateurs.

Un TGT est expiré si la date de dernier usage additionnée de 8 heures (cas.ticket.tgt.primary.time-to-kill-in-seconds) est dépassé ou si la date de création d'un ticket additionnée de 15 jours (cas.ticket.tgt.primary.time-to-kill-in-seconds) est dépassée.

En positionnant le expireAfterSeconds à 0 seconde pour ne plus utiliser le DefaultTicketRegistryCleaner qui prend en compte le lastTimeUsed du TGT (d'où l'obligation de décoder les tickets), la date de dernier usage n'est plus prise en compte, et le TGT expire systématiquement 8 heures après sa création (même si il est régulièrement utilisé) : la session CAS a donc alors une durée de vie maximale de 8 heures.

Il est dommage que le expireAt ne soit pas mis à jour par CAS à chaque usage du TGT, le comportement n'aurait ainsi pas été changé.
On a donc procédé à un Pull Request en espérant que celui-ci soit pris en compte ou du moins suscite des modifications dans les versions futures de CAS. La modification en elle-même n'est pas très importante, aussi il est étonnant que l'implémentation actuelle du ticket registry MongoDB ne corresponde pas à cette logique de expireAt maintenu comme étant la date d'expiration et expireAfterSeconds étant fixé à 0 ... on verra comment ce PR sera traité (et si des subtilités nous ont échappé ...) :
https://github.com/apereo/cas/pull/5719

En attendant, au vu de cette régression, on préfère ici laisser le paramétrage précédent avec le DefaultTicketRegistryCleaner d'actif, mais on ne lui demande de passer que toutes les heures simplement (plutôt que toutes les 2 minutes) pour économiser un peu de ressources serveur ... :

```
cas.ticket.registry.cleaner.schedule.repeat-interval=PT1H
```

Optimisation MongoDB : suppression des index plein texte

Cf le retour de Pascal concernant la consommation CPU de MongoDB via CAS, nous avons supprimé les index plein text et avons ajouté un index simple sur le numéro de ticket de cette façon :

```
db.proxyGrantingTicketsCollection.dropIndex('json_text_type_text_ticketId_text')
db.proxyTicketsCollection.dropIndex('json_text_type_text_ticketId_text')
db.serviceTicketsCollection.dropIndex('json_text_type_text_ticketId_text')
db.ticketGrantingTicketsCollection.dropIndex('json_text_type_text_ticketId_text')

db.proxyGrantingTicketsCollection.createIndex({ ticketId: 1 }, {name:'json_text_type_text_ticketId_text'})
db.proxyTicketsCollection.createIndex({ ticketId: 1 }, {name:'json_text_type_text_ticketId_text'})
db.serviceTicketsCollection.createIndex({ ticketId: 1 }, {name:'json_text_type_text_ticketId_text'})
db.ticketGrantingTicketsCollection.createIndex({ ticketId: 1 }, {name:'json_text_type_text_ticketId_text'})
```

En forçant le nom des nouveaux index simples par le nom des index plein texte créés normalement par CAS, CAS n'écrase pas ces index par défaut.
cas.ticket.registry.mongo.update-indexes=false fait également en sorte plus simplement que ces modification sur ces index ne soient pas écrasés.
La bonne solution serait évidemment que la correction soit apportée dans CAS directement.

Quel Ticket Registry en CAS 6.6.9 ?

Si le Ticket Registry MongoDB donne satisfcation à l'usage, et nous permet d'abandonner le Ticket Registry Redis maintenant inutilisable en CAS 6.6.9, il reste donc dommageable de ne pas mettre à profit les possibilités d'expiration des documents via TTL proposées nativement par MongoDB. Le Ticket Registry MongoDB pourrait être revu et corrigé. On espère que notre PR aidera au moins un peu à aller dans ce sens.

On peut aussi tenter de trouver un Ticket Registry qui s'occupe véritablement nativement et dès maintenant du nettoyage des tickets sans passer par le DefaultTicketRegistryCleaner dont l'implémentation et le coût sont les même pour tous les Ticket Registry quand celui-ci est actif (cf parapgraphe plus haut dans cette page), à savoir : lecture et décodage de tous les tickets de la base à chaque procédure de nettoyage pour déterminer quels tickets doivent être supprimés.

A inspecter le code de CAS de cette 6.6.9, quelquesTicket Registry sembleraient répondre à cette exigence, on les repère via la surcharge du ticketRegistryCleaner qui résulte de la désactivation pure et simple du DefaultTicketRegistryCleaner : cf l'implémentation de la méthode MemcachedTicketRegistryConfiguration.ticketRegistryCleaner() par exemple

```
    @Bean
    @RefreshScope(proxyMode = ScopedProxyMode.DEFAULT)
    public TicketRegistryCleaner ticketRegistryCleaner() {
        return NoOpTicketRegistryCleaner.getInstance();
    }
```

Les Ticket Registry répondant à cette exigence :

- Memcached
- HazelCast
- Couchbase
- CouchDB

Memcached ne propose pas la persistence entre chaque redémarrage.

HazelCast semble (très) complexe voir "overkill" pour cette fonctionnalité, et si il présente une version "community" a priori "libre", la version enterprise payante nous laisse à penser que la version community ne sera pas forcément adéquate, même pour notre besoin relativement simple.

Couchbase est sur le même modèle : initialement libre, une version enterprise est proposée ; le ticket registry couchbase est dépréciée et disparaîtra de CAS dès la version 7 (supprimé du master).

CouchDB est un projet Apache qui semble véritablement libre et aurait pu être un bon candidat, mais il est en fait également dépréciée et disparaîtra dès la version 7 (supprimé du master).