Central Authentication and Authorization Service

– Web Application のための新しい認証システムの試み –

内藤 久資 (Hisashi NAITO)
名古屋大学多元数理科学研究科
Graduate School of Mathematics,
Nagoya University
naito@math.nagoya-u.ac.jp

梶田 将司 (Shoji KAJITA)
名古屋大学情報連携基盤センター
Information Technology Center,
Nagoya University
kajita@nagoya-u.jp

近年のコンピュータネットワークを利用した種々の情報システムでは, 不特定多数のユーザに対して一様な情報を提供するだけでなく, 各ユーザを特化した情報を提供したり, ユーザからの個人情報の入力を求める場面や, システムに蓄積された個人情報の提供を求める状況が頻繁に発生している. さらには, このような情報システムは, 特定のプラットフォームやアプリケーションを仮定するのではなく, 多様な利用者環境に適応したシステムを構築する必要性から, ウェブブラウザを利用した, いわゆるウェブアプリケーション (Web Application) として構築されることが多い. このような情報システムにおけるユーザ認証の方法として, 我々は, Yale 大学で開発された Central Authentication Service (CAS) を元にし, より強力な認証機能を備えた Central Authentication and Authorization Service (CAS^2) の開発を行った.

本稿では, CAS^2 の持つ強力かつ容易に実現可能な Authentication and Authorization Service の解説を行うとともに, 名古屋大学において 2005 年春に実施した CAS^2 を用いた「学務情報システム」（Web による成績入力・履修申請システム）の運用結果を報告する.

1 背景

はじめに, 一般的な情報システムやウェブアプリケーション（以後, 混乱がない限り「アプリケーション」と省略する場合がある）における認証とは何かを解説する. さらに, 大学等における「分散アプリケーション環境」での認証の問題点と, その一つの Solution としての CAS^2 の役割を考察する.

1.1 認証とは

一般に, 情報システムにおける「認証」とは, システムを利用しようとするユーザが正当な利用者であるかどうかを確認する一連の手続きを指し, その手続きは, 以下の “AAA” と呼ばれる 3 段階に分られることが多い.

Authentication (認証) システムを利用しようとするユーザの正当性をユーザを識別する符号（例えば「ユーザ ID」）と, そのユーザのみが知り得ると考えられる識別子（例え

ば「パスワード」）の組を用いてチェックする.

Authorization (認可) ユーザがアクセスしようとするリソースに対してアクセスを許可されているかどうかをチェックする.

Accounting (課金) ユーザがシステムを利用した際のコストを計算する.

これらの認証の概念はUNIXシステムを例にとるとわかりやすい. Authenticationとはログイン時におけるユーザIDとパスワードを用いたユーザ認証であり, Authorizationとは, 各ファイルのユーザに対するアクセス許可属性である. Accountingは, 直接一般ユーザには関連しないが, UNIXシステムにおいては, 各コマンドがどの程度利用されたかの統計情報として収集されていることが多い. 特にAuthenticationのステージにおいては, 有効なユーザIDとパスワードの組を格納したデータベースに対する問い合わせが必要となり, ホスト内に格納された“BSD Flat Database”（古典的な`/etc/passwd`ファイル）, UNIXシステム群で共有される“NIS”の他, 近年では“LDAP”に代表される, 外部ディレクトリデータベースが用いられる. また, それぞれのデータベースを単独で用いるのではなく, 読み出し順序を指定して複数のデータベースを利用したり, サービスごとにデータベースを切り替える“PAM”などのメカニズムが導入されている.

1.2 ウェブアプリケーションにおける認証の問題点

ウェブアプリケーションにおいても, 基本的な「認証」の概念は大きく変ることはない. アプリケーションにおいて（広い意味で）Authenticationを行うには, 以下の方法を用いることが多い.

1. あるURLパス以下の全てのリソースに対して, 認証データ（「ユーザID」と「パスワードの組」など）を用いる.
2. CGI/Servletなどのアプリケーション内部で認証メカニズムを構築する.

前者の方法は, Apache httpd serverを用いた場合には, `httpd.conf`または, 各ディレクトリにおける`.htaccess`での記述が必要となり, 認証データベースとしては, 各ディレクトリ内に個別にデータベースを用意する“Basic Authentication”の他, `mod_ldap`を用いてLDAPデータベースを利用するなどの外部認証も可能である. この方法は, 設定が容易であるという利点を持つが, Authentication以上の機能を提供することは困難である.

一方, 後者の方法では, CGIプログラミングやServletプログラミングの種々の技術を用いることにより, 極めて強力な認証機能を提供することができる. しかし, 一旦, 認証メカニズムの変更やアクセス権限の変更を行おうとすると, プログラムそのものの変更が必要となる場合も多く, プログラムの技術レベルによっては致命的なセキュリティホールを生むことも珍しくない.

また, アプリケーションで用いられるhttpdプロトコルは, 各ページのデータを取得するごとに通信セッションを閉じるというsession at onceなプロトコルであるため, 一連のCGIまたはServletなどでの認証のためには, 最初にAuthenticationを行った情報を, 後のページに対して順に渡していくというセッション管理の技術が必要となる. すなわち, Authenticationを行ったページで取得した認証情報（そのユーザが正当であるという事実や, ユーザIDなどの情報）を次に開くページに対して, 何らかの方法でパラメータとして渡していく必要

がある. 通常, このセッション管理のためには Cookie と隠された (hidden)URL パラメータを用いることが多く, Java Servlet API ではセッション管理のための標準的な手法が用意されている.

このようにアプリケーションにおける認証メカニズムも, 現在では標準的な手法が確立され, 広く用いられているのだが, ここに述べたように, 強力な認証メカニズムを利用しようとすると, 個別のアプリケーション内で解決しなければならない. 例えば Web Shopping Site のような, 大規模ではあるが単一のアプリケーションであったり, ひとつのグループが認証データベースも含めて管理を行うようなサイトの場合には, 該当の管理グループ内で問題を解決することが可能である.

ここで, 大学内の情報システムを例にとって, ここまでに述べた認証メカニズムの問題点を考えてみよう. 大学内における情報システムは, 必ずしも単一の管理グループが管理するシステムではなく, 種々のレベルの管理グループが独立して情報システムの管理を行うことが多い. 例えば, 全学レベルの情報システムだけを対象としても, 履修登録などの学務情報や研究者ディレクトリなどの研究情報などは, 一般には異なった管理グループによって管理されている. さらには, 部局ごとの情報システムは, 各部局によって管理される. このような多様な情報システムが統一した認証情報を用いようとすると, 全学共通の認証データベースに対して, 多様なレベルの情報システムからのアクセス権を保証する必要がでてくる. しかしながら, ウェブアプリケーションによる情報システムは, 外部からのクラッキングに対して脆弱な部分を持つことが考えられる. すなわち, ひとつの情報システムに脆弱な部分が存在し, そこからクラッキングを受けた場合, その情報システムが認証データベースへの十分なアクセス権限を持つ場合には, 認証情報の流出が発生し, 他の情報システムまでその被害がおよぶ可能性がある.

1.3 ウェブアプリケーションに対する統一認証基盤

大学などの多様な管理形態を持つ情報システムを統一した認証情報を用いて管理することは, ユーザサイドからは, 単一のユーザ ID とパスワードを用いてアクセス可能であるという利点を持つが, 一方では, ひとつの情報システムの脆弱性が他のシステムに波及する危険性をあわせ持つこととなる.

これを回避する方法としては, 「ポータルサイト」を用いた “Single Sign On” メカニズムを利用する方法が考えられる. この場合, ポータルサイトのみが認証データベースへのアクセス権を持ち, 種々の情報システムをポータルサイトのひとつの “Channel” として構築する. 各情報システムはポータルサイトが得た認証情報のうちで, システムが必要とする情報のみを引き継ぐことが可能となる. このような方法の場合には, 各情報システムから認証データベースへのアクセスが発生しないという利点を持つ. しかしながら, ポータルサイトの Channel として情報システムを構築するのが, 必ずしも容易ではなく, Authorization に対しては, 各情報システム内部で解決する必要がある.

したがって, 多様な管理階層を持つ情報システムの集合体では, 統一した認証データベースを用いた認証メカニズムが必要不可欠であると考えられ, その統一認証基盤には, 個別の情報システムが認証メカニズムには深くは依存せず, 個別の情報システムのセキュリティレベルの低下が, 他の情報システムに波及しないことが求められる.

本稿では, このような条件をみたす統一認証基盤としての Central Authentication Service とその拡張について, 我々が開発したメカニズムとその運用結果について報告したい.

2 Central Authentication Service とは

前節で考察した統一認証基盤は, 最低でも以下の条件がみたされる必要があると考えられる.

1. 個別の情報システム自身は, 認証データベースへのアクセス権限を持たないこと.
2. 情報システムの構築の際に, 認証部分の記述が容易であり, 他の部分は認証メカニズムに関連しない記述が可能であること.
3. Authentication だけでなく, Authorization およびセッション管理も統一的な手法で管理可能であること.

我々はこれらの条件をみたす認証メカニズムとして Central Authentication and Authorization Service を構築したが, それを解説する前に, その元となった Yale 大学によって開発された Central Authentication Service を考察しよう.

2.1 Central Authentication Service の概要

Yale 大学 ITS Technology & Planning は, 上記の条件のうち 1 および 2 をみたす認証メカニズムを提供するシステムとして Central Authentication Service (CAS) を開発した. (See. [1]) CAS はオープンソースとして開発されているため, 我々はこれを元に統一認証基盤のメカニズムを開発することを考えた. 我々が開発した CAS^2 を解説する前に, CAS^2 のベースとして利用した CAS の基本事項をまとめておこう.

通常, 静的なウェブページまたは CGI 等で認証つきのページを構築しようとする場合の動作は Figure 1 のようになる[1]. すなわち, アプリケーション自身が認証サーバにアクセスする必要があり, 次のような欠点が考えられる.

1. アプリケーションに認証サーバへアクセスするためのコードを記述する必要がある.
2. アプリケーションが認証サーバにアクセスする権限を持たなければならない.
3. 同一のブラウザから他のアプリケーションにアクセスするごとに認証を受けなければならない.

CAS はこれらの欠点を補うために考え出されたシステムで, Java Servlet API 2.3 を用いて構築されている[2]. CAS は, 他の情報システムとは独立し, 認証機構のみを担当するアプリケーションとして動作する. また, それ自身が認証データベースを持つのではなく, 外部の種々の認証データベース（例えば LDAP サーバ, Oracle データベースサーバなど）への問い合わせとその結果の送信のみを担当する. CAS を導入した場合, CAS 認証つきのアプリケーションにアクセスする際の基本的な動作は Figure 2 の通りとなる.

[1] Figure 1 では, 認証サーバとして, アプリケーションサーバの外部にある LDAP サーバを参照することを仮定している.

[2] したがって Tomcat などの上で動作する.

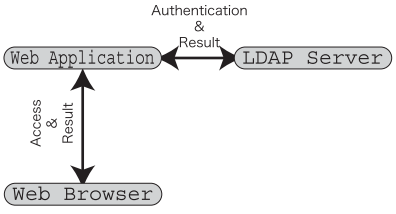

Figure 1

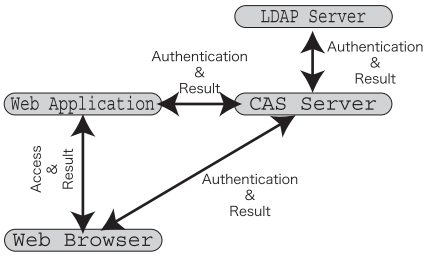

Figure 2

2.1.1 CAS 認証 (1)

実際の CAS 認証つきのアプリケーションへのアクセス時の動きは以下のようになる.

1. CAS 認証つきのアプリケーションにアクセスの際に, アプリケーションは, “Service Ticket”（以下では “ST” と書く）と呼ばれる URL パラメータをチェックする. (**Figure 3**-1, p.7)
2. ST が存在しない場合, または, ST が無効である場合[3], アプリケーションは, CAS サーバへのリダイレクションを発生し, ブラウザから CAS サーバへのアクセスが行われる. (**Figure 3**-2a)
3. CAS サーバは, ブラウザから渡される “Ticket Validation Cookie”（以下では “TGC” と書く）と呼ばれる, クッキーをチェックする. その際, リダイレクションによる CAS サーバへのアクセスには, “Service Parameter” が URL パラメータとして CAS サーバへ送付される. Service Parameter とは, 「どの URL(アプリケーション) へアクセスしようとしているか」を示すパラメータである.
4. TGC が存在しない場合には, ブラウザに対して「ログインウィンドウ」のデータを送付 (**Figure 3**-2b) し, ブラウザからの「ユーザ認証データ」を受け取り, ユーザ認証を行う. ユーザ認証に成功した場合には, ブラウザに対して, 有効な TGC を送付する. (**Figure 4**-3 p.7)
5. TGC が存在する場合, 及び, ユーザ認証に成功し, TGC を送付するときには, CAS サーバは, 有効な ST を生成し, それを URL パラメータに含むアプリケーションへのリダイレクションを発生する. 同時に, CAS サーバは, 生成された ST を内部データベースに保存する. その際, ST データベースには「アクセスしているユーザ名」を ST の値をキーとしてデータベース内に保存する. (**Figure 4**-4)
6. アプリケーションへのアクセスが, ST を URL パラメータに含む場合には, アプリケーションは, ブラウザから受け取った ST を CAS サーバに送付する. (**Figure 5**-5a, p.7)
7. アプリケーションから ST を受け取った CAS サーバは, アプリケーションに対して ST の有効性と（有効な場合には）ユーザ名などの ST データベースに保存されているデータを送付する. (**Figure 5**-5b)
8. アプリケーションは, CAS サーバから受け取った ST の有効性の結果を元に, アクセス権を持つユーザに対しては, ブラウザに対して適切なレスポンス（Web ページコンテンツ）を送付する. (**Figure 5**-6)

[3] ST の有効性は, ステージ 6 でも検証される.

上のアクセスの様子をより詳細に述べると, 以下のようなアクセスが発生する. ここではアプリケーションの URL を **`https://foo.nagoya-u.ac.jp/app/`**, CAS サーバの URL を **`https://cas.nagoya-u.ac.jp/`**とする.

1. アプリケーションの (パラメータつき) URL が

   ```
   https://foo.nagoya-u.ac.jp/app/?param1=value1&param2=value2
   ```

   である場合, ST を付けた URL は

   ```
   https://foo.nagoya-u.ac.jp/app/?ticket=ST-XXXXXX
    %3Fparam1=value1%26param2=value2
   ```

   となる. この URL 内にあらわれる **`ticket`** パラメータの値が ST であり, **`XXXXXX`** の部分はランダムに生成されている. (**Figure 3**-1)
2. ST が存在しない場合, または無効な場合に発生するリダイレクションは Java Script を利用したものであり, リダイレクション先は

   ```
   https://cas.nagoya-u.ac.jp/index.jsp
    ?service=//foo.nagoya-u.ac.jp/app/
    %3Fparam1=value1%26param2=value2
   ```

   となる. (**Figure 3**-2a)
3. CAS サーバがブラウザに対して以下の情報を持つクッキー (TGC) を発行する.

   - “パス” **`cas.nagoya-u.ac.jp`**
   - “クッキー文字列” **`TGC-XXXXXX`**
   - “セキュアフラグ” “ON”

   ここで,「セキュアフラグ」が立っているクッキーとは, SSL 通信の場合にのみ, ブラウザがクッキーをサーバに送付することを条件付けられたことを意味する.
4. 認証に成功した場合には, 3 で示した TGC をブラウザに送付する. (**Figure 4**-3)
5. CAS サーバから発生する Java Script によるリダイレクションは

   ```
   https://foo.nagoya-u.ac.jp/app/?ticket=ST-XXXXX
    &service=https://foo.nagoya-u.ac.jp/app/
    %3Fparam1=value1%26param2=value2
   ```

   となる. (**Figure 4**-4)
6. アプリケーションはブラウザから受け取った ST を CAS サーバに

   ```
   https://cas.nagoya-u.ac.jp/Validate/?ticket=ST-XXXXX
   ```

   として送付する. (**Figure 5**-5a, p.7)
7. 上記 6 のアクセスを受け取った CAS サーバは, **`ticket`** パラメータとして含まれる ST を検証する. (**Figure 5**-5b)
8. CAS サーバはアプリケーションに対して, ST の検証結果を **Figure 6** の形の XML ファイルとして送付する[4]. この結果を受け取ったアプリケーション (CAS クライアント) は, XML コードから **`netid`**(ユーザ ID) を読み取ることができる. (**Figure 5**-6)

[4] この XML コードは ST の検証に成功した例であり, 一部を省略している.

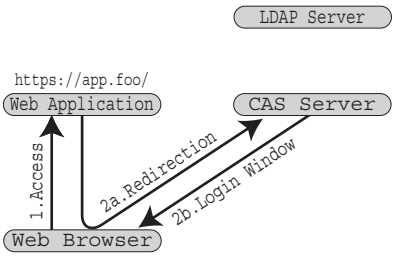

Figure 3: CAS の動作 (1)

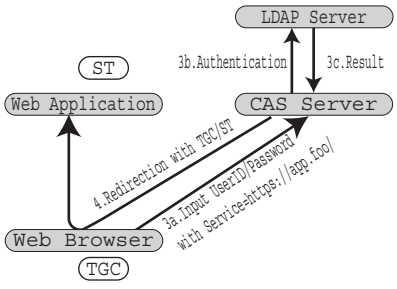

Figure 4: CAS の動作 (2)

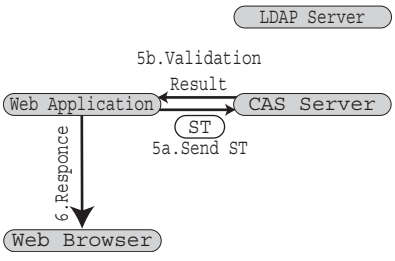

Figure 5: CAS の動作 (3)

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

Figure 6: CAS が返す XML データ

また, CAS サーバ内部における TGC/ST データベースは, 以下に示す構造で格納されている.

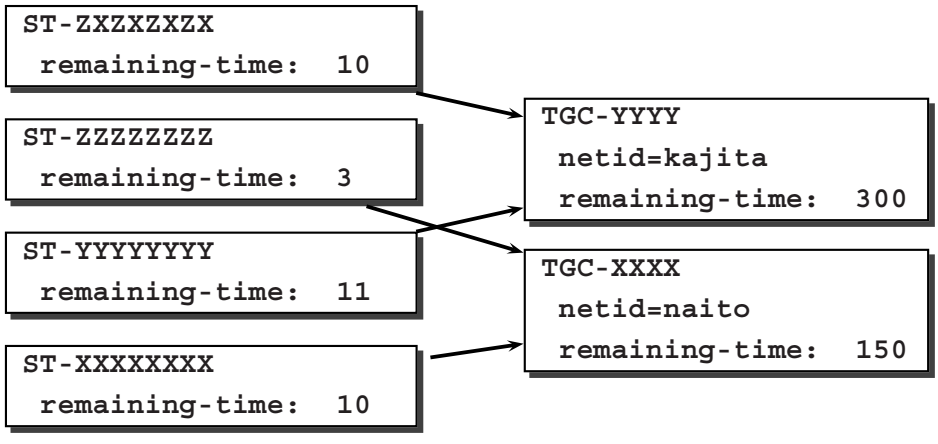

Figure 7: CAS サーバ内部におけるデータベース構造

2.1.2 CAS の認証の基本要素

一見複雑に見える CAS サーバを用いた認証方法であるが, 認証の基本となる概念は次の２つである.

★ **Ticket Granting Cookie** ブラウザ（ユーザ）が認証済みかどうかを判断するためのクッキーであり, ブラウザ内に “Session Once” で保存される[5]. また, 発行された TGCd は

[5]“Session Once” クッキーとは, ブラウザを閉じたときに破棄されることを意味する. TGC を破棄することは, CAS サーバからのログアウトを意味する.

CAS サーバ内のデータベースに保存される. ブラウザと CAS サーバ間のみで交換される.

★ **Service Ticket** アプリケーションへのアクセスが, 認証されたブラウザ（ユーザ）からのアクセスであるかどうかを判断するための URL パラメータであり, CAS サーバ内のデータベースに保存される[6]. ブラウザからアプリケーションへの “One Time Ticket” の役割を果たし, 一旦利用された ST は破棄される.

この CAS の動作機構からわかるように, CAS は基本的な 2 つのサーブレットを持つ.

★ **Login Servlet** ブラウザからのアクセスを受理し, ブラウザの TGC のチェックまたは, ブラウザに対する TGC の配布を行う. また, ブラウザに対して新規 ST を発行する.

★ **Validation Servlet** アプリケーションからのアクセスを受理し, ST のチェックとアプリケーションに対する認証結果の送信を行う.

また, 付随的なサーブレットとして次のものを持つ.

★ **Logout Servlet** ブラウザからのアクセスを受理し, ブラウザの TGC を破棄する. すなわち「ログアウト」操作を行う.

これら, TGC 及び ST は CAS サーバ内では「有効期限」を持つ. TGC の有効期限は, ある程度長く設定され[7]一種の “Session Timeout” の役割を果たす. 一方, ST の有効期限は短い値に設定することが必要となる[8]. これは, ブラウザから CAS サーバへのアクセスに際しては, TGC を用いないため, ST が流出すると Man-in-Middle Attack の対象となるためである.

ここに述べたように, CAS では, クッキー, URL パラメータ, (Java Script) リダイレクションなどの Web における標準的とされる技術のみが利用されているため, ほとんどのシステムで CAS が利用可能となる.

2.1.3 CAS 認証 (2)

また, 同一の CAS サーバを利用する, 複数のアプリケーションを利用する場合では, どれか一つのアプリケーション上で「ログイン手続き」を行えば, ブラウザ上には CAS サーバに対する TGC が保存されるため, 他のアプリケーションにアクセスする際に, 再度ログイン手続きを行う必要はない. このような意味で, 同一の CAS サーバを利用するアプリケーションに対しての “Single Sign On” が実現できる.

なお, これらのブラウザやアプリケーションと CAS サーバ間の通信を安全に維持するためには, 全ての通信が SSL Layer 上で行われる必要があるが, CAS サーバが行う全ての通信はウェブベースであるため, `https` 通信を用いることにより, 容易に暗号化通信が実現できる.

---

[6] ST データベースの属性値として TGC が格納され, TGC データベースの属性値として「ユーザ情報」が格納される.

[7] 例えば 1 時間程度. CAS サーバは, TGC を持ったブラウザからのアクセスがあった時点で, TGC のカウントダウンタイマを更新する.

[8] 例えば 1 分程度.

2.2 Central Authentication Service への対応方法

このように, CAS を用いることにより, アプリケーション自身がユーザ認証を行わずに済む利点があるが, アプリケーションを CAS を利用した認証に変更すること[9]に多くのコストがかかるのであれば, アプリケーション自身がユーザ認証機構を組み込んだ方が安価となってしまう. 既存のアプリケーションを CAS 化することは容易であり, 実際の CAS 化のためのアプリケーションコードの変更方法を簡単に解説しよう.

ここでは, 簡単のため, 次のような CGI によるアプリケーションを構築してみよう.

1. “Welcome Page” の中に「ユーザ ID とパスワード」の入力欄がある.
2. 認証に成功すると「ユーザ ID」を表示する.

これは, 認証つきでセッション維持を行わないアプリケーションの最も単純なもので, これを実現する CGI コードは以下のようになっていると考えられる.

```
### FORM 入力の decode （ユーザ ID/パスワードのデコード）

### ユーザ ID, パスワードを利用して認証サーバと通信
### 認証結果, ユーザ情報などを得る

### HTML データ生成 （ユーザ ID の表示）
```

このように記述された CGI コードは, 以下のように書き換えるだけで CAS 化することができる.

```
### FORM 入力の decode （ユーザ ID/パスワードのデコード）

### CAS client ライブラリの呼び出し
### 認証結果, ユーザ情報などを得る

### HTML データ生成 （ユーザ ID の表示）
```

すなわち, CGI などの中で認証サーバと通信している部分を「CAS クライアントの呼び出し」に置き換えるだけでよい. CAS クライアントは, すでに Yale 大学などで, perl, Java, PHP, PL/SQL, Python, Ruby など, CGI やサーブレットを作成するために利用されている多くのプログラム言語に対して開発されている.

2.3 問題点

このように, CAS はアプリケーションに対する強力な認証機構を提供しているが, Yale 大学が提供している CAS 自身にはいくつかの問題点があることが知られている. そのうちの代表的なものを挙げておこう.

★ Form 入力のメソッドのうち GET メソッドにしか対応していない

Form 入力での GET メソッドは, パラメータを含む URL の文字数に制限があるだけでなく, 入力データが URL パラメータとして表示されてしまうため, セキュリティを保つ必要のある Form 入力で GET メソッドを利用することはできず, 通常は POST メ

[9] “**CASify**” または “**CAS 化**” と呼ぼう. CAS を用いるように変更されたアプリケーションを “**CASified** Application” と呼ぶ.

ソッドを用いる. しかしながら, 以下の理由により CAS は POST メソッドには対応できていない.

あるアプリケーションに対して POST メソッドによる Form 入力を行ったとしよう. また, その Form 入力時に, 有効な ST が存在しない状況を考えよう. このときの動作は以下のようになる.

1. Form 入力を受け取ったアプリケーション上では, 最初に CAS クライアントライブラリが呼び出され, ST のチェックが行われる. しかし, その ST が有効でないため, ブラウザに対して CAS サーバへの Java Script によるリダイレクションが発生する. この際,CGI スクリプトの実行は, CAS クライアント呼び出しの時点で終了する.
2. ブラウザが TGC を持っていれば CAS サーバから有効な ST を含む, アプリケーションに対する Java Script によるリダイレクションが発生する.

この一連のアクセスで Form 入力が正しく行われるためには, 2 のリダイレクション内に, 全ての Form データが含まれる必要がある. しかしながら, CAS で利用される, リダイレクションのための JSP が GET メソッドにしか対応できていないだけでなく, CAS のサーブレット内部においても, アクセスのための Service Parameter のみが JSP に書き込むため, CAS を POST メソッドで利用すると 1 のリダイレクションにおいて, POST された Form データが消失してしまう.

★ 国際化に対応していない

Form 入力で用いられる文字コード体系は, その Form を含む HTML ファイルの文字コード体系が用いられる. すなわち,Form 入力を行うウェブページの文字コード体系が EUC-JP であれば, そこで送信される Form データも EUC-JP でエンコードされる. 一方 CAS は Java を用いて構築されているため, CAS の内部コードは UTF-8 となり, EUC-JP でエンコードされた URL パラメータまたは Form データを正しくリダイレクションできない[10].

★ CAS サーバへのアクセスを制限する機構を持たない

CAS を構成するサーブレットのうち, ブラウザからのアクセスを受理する Login は, 任意のブラウザからのアクセスを受理する必要がある. しかし, アプリケーションからのアクセスを受理する Validation が, 任意のアプリケーションからのアクセスを受理した場合, ランダムな ST を CAS サーバに大量に送付することにより, 有効な ST やそれに付随するユーザ情報が流出する可能性がある.

★ Cross Site Scripting 脆弱性が存在する

CAS サーバに対して以下の URL を送信してみよう.

```
https://mynu.jp/cas/index.jsp?service=javascript
%3aalert%28document.cookie%29%3b
```

これは, 典型的な Cross Site Scripting を行う URL であり, これによって TGC が流出することがわかる[11].

[10] もちろん, アプリケーションを UTF-8 で記述すれば, この問題とは無関係となる. しかし, データが EUC-JP や Shift-JIS などで記述された Oracle データベースを backend に持つようなシステムでは, PL/SQL スクリプト自体をデータのエンコーディングと一致させる必要があるため, スクリプトを UTF-8 に変更することは容易ではない.

[11] 名古屋大学情報連携基盤センタープロジェクト研究員の杉浦達樹氏によって発見された.

これらの問題点のうち, 前者 2 つは CAS サーバ内部のパラメータ処理の問題である. 一方, 後者 2 つは, CAS サーバが Authentication のみを行い, アプリケーションへのアクセス権限管理 (Authorization) を行っていないことに原因がある.

これらの問題点をクリアすることは, 実用に耐える日本語を用いたアプリケーションをセキュアに運用するには必要不可欠な要素であり, 特に Validation への無制限なアクセスが許されることは, 極めて重大な情報流出につながる危険性がある.

3 Central Authentication and Authorization Service とは

我々が考える統一的な認証基盤は, 多様な管理階層によるアプリケーションのセキュリティと高可用性をサポートするものでなければならない. 統一認証基盤として CAS を利用するだけでは, 前節の問題点があるだけでなく, 各アプリケーションに対してアクセス権を持つユーザの設定・クライアントの制限・アクセス時間帯の制限など, 様々な権限管理機構 (Authorization 機構) が必要になる. このような権限管理機構をアプリケーション側に組み込むためには, アプリケーションは, ユーザ ID 以外の種々のユーザ情報を CAS から受け取る必要があり, 「ユーザ認証情報は CAS のみが扱う」という基本ポリシに反する場面が生じる. また, アプリケーションが必要とする最小限のユーザ情報へのアクセス権限を認証データベース上で設定しようとすると, 認証データベース上に, 各アプリケーションに対する権限管理情報を設定する必要が生じる. このような状況は, アプリケーション, 認証サーバ, 認証データベースの階層構造を壊すものであり, システム管理上好ましい設定ではない.

我々は, このような権限管理機構を CAS に組み込み, アプリケーションのセキュリティと高可用性をサポートすることに成功した. 実際, 我々が行った権限管理機構は, “Service Based Authorization” と考えられるもので, CAS サーバ (Validation) 内で Service パラメータをキーとした Authorization を行うことである. この機構の導入により, CAS サーバへのアクセス制限だけでなく, Cross Site Scripting 脆弱性も同時に克服することができる.

以下では “Service Based Authorization” に基づく権限管理機構を組み込んだ CAS, すなわち CAS^2 の権限管理機構とその管理方法などについて解説する.

3.1 CAS^2 での権限管理データーベース

CAS^2 での Authorization は, **`service`** パラメータを権限管理の鍵とする認証体系であるため, この考え方を “Service Based Authorization” と呼ぶこととする. この **`service`** を鍵とする Authorization は Validation サーブレットで行い, アプリケーションから Validation サーブレットに渡された **`service`** パラメータを CAS^2 が持つ権限管理データベースと照合する.

CAS^2 においては, 権限管理データベースは, 認証データベースと同じく, 外部データベースを利用する. 以下では, この権限管理データベースとして, LDAP ディレクトリサーバを用いていると仮定して議論を進める.

権限管理データベースにしたがって, 我々は以下の意味での権限管理機構を導入した.

- ST の検証 (Validation) 要求を受理した際, アクセスを行おうとするユーザが該当の URL に対してアクセス権を持つかどうか.

- アクセス権がある場合に, アプリケーションに送信するユーザ情報を任意に設定可能とする.

この権限管理機構を実現するアクセス権限リストを “CAS Access Control List” (CAS-ACL) と呼ぶ. 以下では CAS-ACL の役割と記述方法を, CAS-ACL が LDAP サーバに格納されていると仮定して解説する[12].

3.1.1 Access Control List

CAS-ACL は, 以下のように記述される.

- 標準的な CAS-ACL エントリ：

```
dn: cn=uPortal,ou=cas,o=NU
cas-auth-type: basic
cas-attributes: uid,MailAddrss,username,dn
cas-service: https://app\.foo/.*
cas-allow: (dn=.+,ou=people,o=nu)
```

この CAS-ACL エントリがあらわすアクセス権限は以下のようなものである.

- 対象となる URL は **cas-service** に記述された正規表現にマッチする URL であり, この場合には, **https://app\.foo/.*** にマッチする URL が対象となる.
- URL が **cas-service** の値にマッチしたとき, アクセスが許されるユーザは, そのユーザの LDAP エントリが **cas-allow** に記述された正規表現にマッチするユーザに限る. この場合には, ユーザエントリの **dn** の値が **dn=.+,ou=people,o=nu** にマッチしたときとなる.
- 上記の Authorization によりアクセスが許可されたとき, **cas-attributes** に記述された情報のみがアプリケーションにユーザ情報として渡される. この場合には, ユーザの LDAP エントリでの **dn** の値, 及び, **uid**, **MailAddress**, **username** の各属性値がアプリケーションに渡される.

この CAS-ACL エントリの構造から, 自然に「正規表現で記述された URL のグループ」が構成され, これを “CAS Access Control Class” (“CAS-ACC”) と呼ぶ.

アプリケーションからの検証要求を受理した Validation は, **service** パラメータにしたがって, それと一致する **cas-service** 属性値を持つ CAS-ACL を検索し, アクセス権限の検証 (**cas-allow** 属性値との一致) を行う. さらに, アプリケーションへ送信するユーザ情報の属性値を **cas-attributes** エントリから読み取る. （実際には, ST 発行時にも CAS-ACL との照合を行う. 詳細は 3.3 節を参照.）

3.1.2 Access Control List によるアクセス制御の実例

CAS-ACL の **cas-allow** 属性値は, LDAP の任意の属性名と属性値に関するマッチングを表現できるほかに, アクセス可能日時・時間帯・アクセス元の IP アドレスなどを指定す

[12] 他のデータベース形式であっても, 属性名と属性値の組合わせが同等であれば, CAS[2] のデータベース読み出しルーチンを変更するだけで対応可能である.

ることも可能で, それらの任意の組合わせを利用可能である. また, **cas-attributes** 属性値には, ユーザに関する任意の LDAP の属性値を以下のようにして指定可能である.

- アクセス時間帯を午前 9 時から午後 5 時に限るための記述.

```
cas-allow: (&(time>=0900)(time<1700))
```

- アクセス日時を 2005 年 7 月 1 日から 2005 年 7 月 31 日に限るための記述.

```
cas-allow: (&(date>=20050701)(date<=20050731))
```

- アクセス日時を 2005 年 7 月 1 日午前 9 時からから 2005 年 7 月 31 日午後 5 時に限るための記述.

```
cas-allow: (&(datetime>=200507010900)(date<=200507311700))
```

- アクセスを月曜日から金曜日に限るための記述.

```
cas-allow: (&(wday>=Mon)(wday<=Fri))
```

- アクセスは「名古屋大学内部」からに限る.

```
cas-allow: (IP=133.6.0.0/16)
```

これらのアクセス時間帯制限などの記述は, 「逆ポーランド式」の論理式によって記述可能であるが, 実際の運用は複雑なアクセス制限の記述が必要である. その例として, 以下のようなアクセス制限を考えてみよう.

- 基本的なアクセス制限は, 「2005 年 7 月 1 日午前 9 時から 2005 年 7 月 31 日午後 5 時まで」である.
- ただし, 「毎日午前 3 時から午前 5 時」は保守のためアクセスを禁止する.
- アクセスは「名古屋大学内部から」に限る.
- アクセスは「教員」に限る. 以下では, 「教員」を識別する正規表現として,

  ```
  dn=.+,ou=staff,ou=people,ou=NU
  ```

  を利用できると仮定する.

このアクセス制限を実現する **cas-allow** の属性値は, これをそのまま記述すれば,

```
(&(&(dn=.+,ou=staff,ou=people,ou=NU)
  (&(datetime>=200507010900)(date<=200507311700)))
  (&(IP=133.6.0.0/16)(|(time>0300)(time<=0500))))
```

と, 極めて冗長なものとなり, 誤った記述を引き起こす原因となる.

このような冗長な記述を回避するため, 我々は CAS-ACL に「マクロ定義」に相当するエントリを記述可能とした.

- 「マクロ」CAS-ACL エントリ：

```
dn: cn=access_time,ou=cas,o=NU
cas-auth-type: accessfilter
cas-allow: (&(datetime>=200507010900)(date<=200507311700))

dn: cn=without_mentenance_time,ou=cas,o=NU
cas-auth-type: access_filter
cas-allow: (|(time>0300)(time<=0500))
```

これらの「マクロエントリ」は **cas-auth-type: access_filter** で特徴づけられ, 上記の「冗長」な論理式は,

```
(&(&(dn=.+,ou=staff,ou=people,ou=NU)
  (access_filter=cn=access_time,ou=cas,o=NU)
  (&(IP=133.6.0.0/16)
  (access_filter=cn=without_mentenance_time,ou=cas,o=NU))))
```

と記述することができる. さらに,

```
dn: cn=access_time_0,ou=cas,o=NU
cas-auth-type: access_filter
cas-allow: (&(access_filter=cn=access_time,ou=cas,o=NU)
  (access_filter=cn=without_mentenance_time,ou=cas,o=NU))

dn: cn=staff_in_univ,ou=cas,o=NU
cas-auth-type: access_filter
cas-allow: (&(dn=.+,ou=staff,ou=people,ou=NU)
  (IP=133.6.0.0/16))
```

と定義しておけば,

```
(&(access_filter=staff_in_univ,ou=cas,o=NU)
  (access_filter=cn=access_time_0,ou=cas,o=NU))
```

まで短縮することが可能となる. もちろん, そのマッチング処理のためには, それぞれのエントリの呼び出しと各エントリでのマッチングを行う必要がある.

3.2 Service Ticket の発行方法の変更

我々は権限管理機構の導入と同時に, ST の発行方法も変更した. ST はアプリケーションへのアクセスのための “One-Time Ticket” の役割を果たすため, 一旦利用した ST は再度利用されることはない. したがって, 同一ページまたは他のページへのアクセス時には, 必ず Login サーブレットへのリダイレクションが発生する. 具体的には, 一つのページへのアクセスのためには, 3 回のブラウザと CAS サーバまたはアプリケーションとの通信が必要となる.

このアクセス回数を減少させるため, 我々は Validation サーブレットからアプリケーション経由で, ブラウザに「次回, 同一の CAS-ACC に属する URL にアクセスする際に有効なチケット」を発行するように変更した. このサービスチケット（ST）を “nextticket” と呼ぶ. これによって, 同一の CAS-ACC に属する URL へのアクセスに際しての通信回数は 1 回に減少することとなる. また, 各 CAS-ACC に対して nextticket を発行するか否かは, CAS-ACL の **cas-attributes** の属性値で指定可能とした[13][14]. CAS[2] サーバ内部では, 以下のように ST/TGC データベースが保持されている.

[13]実際には nextticket を発行することがデフォールトであり, **cas-attributes** に **noNextTicket** を指定することにより, nextticket を発行しない設定にすることができる.

[14]次節（3.3 節）では, nextticket に関するセキュリティについても議論する.

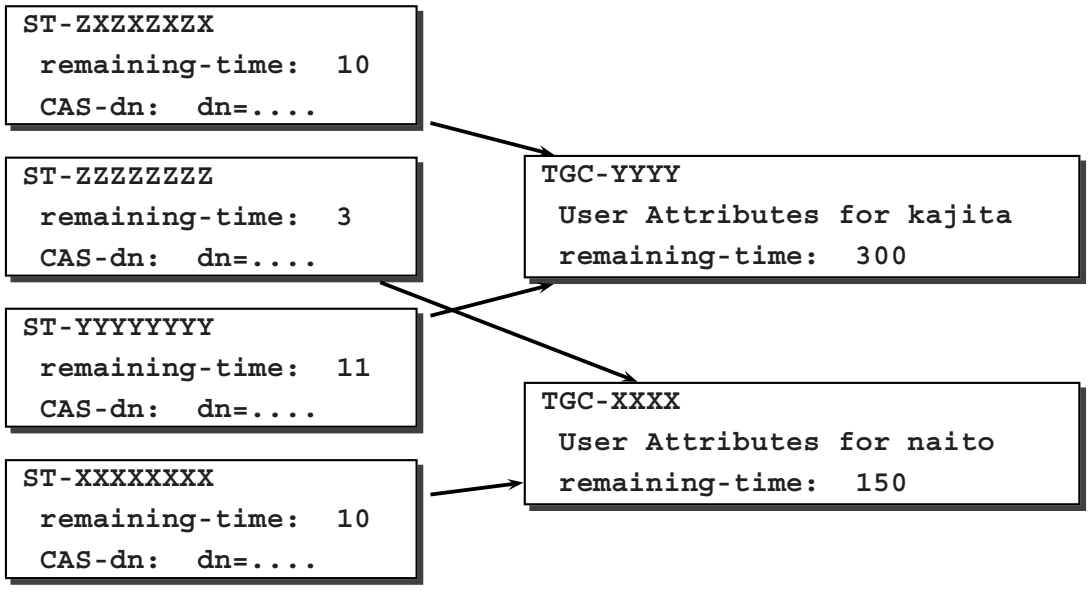

Figure 8: CAS^2 サーバ内部におけるデータベース構造

ここで, TGC データ内の “**User Attributes**” は, 認証データベースから得られた全属性値[15]を「Hash Table」として保持している部分である. したがって, CAS-ACL で指定された, 任意の属性値を（その属性が存在している限り）TGC データベースから読み出すことができる.

また, Validation からブラウザに渡される ST の認証結果の XML コードは以下のように変更した. ここで, <cas:Attributes>以下は, CAS-ACL で指定された属性名と属性値の組である.

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
  <cas:authenticationSuccess>
    <cas:ticket>ST-XXXXX</cas:ticket>
    <cas:user>netid</cas:user>
    <cas:Attributes>
      <cas:attribute-1>attribute-1-value</cas:attribute-1>
      <cas:attribute-2>attribute-2-value-1,
        attribute-2-value-2</cas:attribute-2>
    </cas:Attributes>
  </cas:authenticationSuccess>
</cas:serviceResponse>
```

Figure 9: CAS^2 が返す XML データ

CAS(または CAS^2) クライアントは **Attributes** 以下の XML データを「Hash Table」としてデコードすれば, その認証結果を保存する変数を “**result**” としたとき, これまで, UserID を “**userid=result.netid**” として読み出していたが, 他の属性値は

[15]認証データベースから読み出しを許された全属性値であり, 一般にはパスワードデータは読み出すことができない.

“**fullname=result.attributes.fullname**” などとして読み出すことが可能となる[16]

3.3 認証メカニズム

アクセス権限機構と nextticket の導入により, 従来の CAS と比較して, 認証メカニズムに多少の違いが生じる.

3.3.1 CAS^2 認証 (1)

まず, 2.1.1 節と同じく, CAS 認証を必要とするアプリケーションに, ユーザがはじめてアクセスする際の動作過程を 2.1.1 節と比較しながら解説する.

1. アプリケーション（例えば **https://foo.nagoya-u.ac.jp/app/**）に ST なしにアクセスする. このとき, アプリケーション（CAS クライアントライブラリ）から Login サーブレットへのリダイレクションによって「ログイン画面」が提示される. この過程は 2.1.1 節 (1) と同一である.
2. 2.1.1 節 (3) と同じく, CAS サーバ（Login）は認証を行い, ブラウザに対して TGC を発行する. ここで, ST を発行する際に, **service** パラメータおよびユーザ情報等を用いて CAS-ACL との照合を行う. (Figure 10 参照) CAS-ACL を用いたアクセス権限の照合 (Authorization) に成功したときのみ ST を発行し, リダイレクションを発生する. また, CAS^2 サーバ内の ST データベースには, 該当した CAS-ACC (CAS-ACL の **dn** の値) が保存される.
3. 2.1.1 節 (4) と同じく, ST アクセスを受理したアプリケーション（CAS クライアント）は, CAS サーバ (Validation) に

   ```
   https://cas.nagoya-u.ac.jp/Validate/?ticket=ST-XXXXX
    &service=https://foo.nagoya-u.ac.jp/app/
    %3Fparam1=value1%26param2=value2
   ```

   として **ticket** パラメータだけでなく, **service** パラメータも送付し, ST の正当性・有効性をチェックする. ここでの正当性には, アクセスされた URL から得られる CAS-ACL に基づくアクセス権限の照合のみならず, ST データベースに保存された CAS-ACC と, この段階で得られた CAS-ACC との照合を行う. これら, ST の正当性・有効性が確認できた場合に, 該当のユーザ情報をアプリケーションに送付する. その際に送付するユーザ情報は CAS-ACL で指定されたユーザ情報のみである. また, CAS-ACL において nextticket の送付が指定されている場合には, 新規に生成した ST をアプリケーション経由でユーザに送付する. 新規 ST は, 旧 ST と同一の CAS-ACC に対応するものとして生成する. (Figure 11 参照)

以上の過程では, ST を発行する段階と, ST の照合を行う段階の 2 回にわたって “Service Based Authorization” が行われる. 重要な検証過程は後者であり, 前者で CAS^2 サーバ内部に

[16] 実際の正しいコードは CAS^2 クライアントの言語にも依存するが, 本質的には同じである.

保存されたSTデータベースのCAS-ACC値と後者での照合を行うことにより, **service**パラメータの改竄による不正アクセス（許可されていないURLへのアクセス：Man-in-Middle attack）を防止することができる.

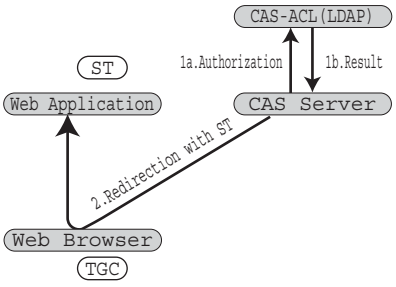

Figure 10: CAS^2 の動作 (1)

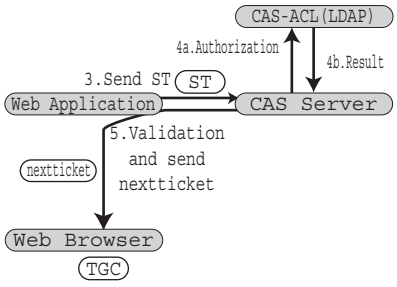

Figure 11: CAS^2 の動作 (2)

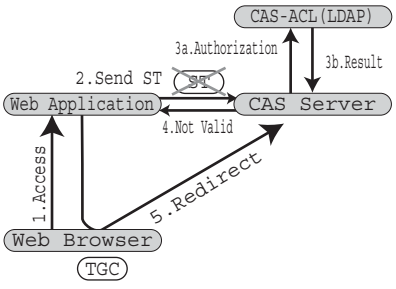

Figure 12: CAS^2 の動作 (3)

3.3.2 CAS^2 認証 (2)

次に, nextticketとして発行されたSTを持つブラウザが前回アクセスしたURLと同一のCAS-ACCに属するURLへアクセスする場合を考えてみる[17].

この場合に, ブラウザからアプリケーションへのアクセスにおいて有効なSTが含まれているため, 2.1.3節で示したLoginサーブレットへのアクセスは発生せず, アプリケーション（CASクライアントライブラリ）からValidationサーブレットへのアクセスでSTの有効性が検証される. このアクセス方式では, ブラウザとCASサーバまたはアプリケーション間の通信回数は1回に減少していることがわかる.

一方, 異なったCAS-ACCに属するURLへのアクセスを行う場合には, nextticketとして入手した新規STの検証の際に, ST発行時点でのCAS-ACCと, アクセスされたCAS-ACCが異なるため, STの検証が失敗する. したがって, STの検証に失敗したアプリケーション（CASクライアント）は, Loginサーブレットへのリダイレクションを生成し, ブラウザに対して, 新規STの取得を求める. (Figure 12参照) しかし, この場合においてもTGC自身が有効である限り, ユーザ（ブラウザ）に対して「ログイン画面」の提示は行われないため, 従来のCAS認証過程と同様にシングルサインオン環境は維持されている.

[17]Figure 9で示した, Validationサーブレットが返すXMLコードの中で, `<cas:ticket>`で示されるものがnextticketである.

3.3.3 Cross Site Scripting に対する対応

このように “Service Based Authorization” を導入すると, Cross Site Scripting (XSS) 脆弱性への対応が同時に行われる. 通常, XSS は Form 入力フィールドに HTML 構文のタグを送り込むことで行われる. したがって, 入力フィールドから入力された文字列の (HTML 構文における)「特殊文字」の「無害化」(Sanitalize) を行えば, XSS を避けることができる.

CAS サーバに対する XSS は, **`service`** パラメータにスクリプティング文字列を埋め込むことで実現できるが[18], Service Based Authorization を行うことにより, **`service`** パラメータの値（文字列）は, 全て権限管理ルーチンによって処理されるため, そこに埋め込まれたスクリプティング文字列は, いかなる CAS-ACC とも一致しないため, アクセスが拒否されることになる.

3.4 その他の改良

CAS-ACL に基づく Service Based Authorization の他に, 我々は以下のような機能を CAS^2 で実現した.

- POST メソッドへの対応および国際化対応.
- 二重ログインの防止.
- ユーザを一意に識別可能な任意のキーによるログイン機能.

このうち,「二重ログインの防止」は, TGC の新規発行時に, 既存の TGC を検索し, 同一のユーザによる他のブラウザからのログイン状況を把握することで実現した.

前節の CAS^2 認証の過程は, 従来の CAS クライアントで実現可能である. しかしながら, 上記の POST メソッドへの対応および国際化は, CAS サーバの改良および CAS で利用されていたリダイレクトのための Java Script の改良が必要となる. そのため, 我々はブラウザおよびアプリケーションと CAS^2 サーバ間で通信されるパラメータに以下の事項を追加した.

- **`CASREQUESTMETHOD`** パラメータ：アプリケーションへのリクエストメソッドが GET または POST のいずれかを示すパラメータ. デフォールト値は **`GET`** とした. これにより, リダイレクトに用いる Java Script を切り替える.
- **`ENCODING`** パラメータ：アプリケーションが要求している **`character encoding`** の値を示すパラメータ. デフォールト値は **`UTF-8`** とした. これにより, CAS^2 サーバ内部でのパラメータ値のエンコーディングおよび, リダイレクトに用いる Java Script を含む JSP の **`character encoding`** を変更する.

これらのパラメータを利用することにより, 入力メソッドの適切な選択と国際化に対応した.

具体的には, CAS クライアントに対して, それを利用しているアプリケーションが使用する Form 入力メソッド (GET または POST), およびアプリケーションが使用している **`character encoding`** の値を渡す. これらのパラメータをつけたリクエストを受け取った CAS サーバは, それ自身が発生するリダイレクションの中に常にこれらのパラメータを埋めこむ. さらに, Form 入力メソッドが POST の時には, それに対応した Java Script を用いて, 全てのパラメータを POST メソッドにより受け渡しを行う.

[18] CAS サーバに送付する **`service`** パラメータは, Form 入力 (GET メソッド) による入力と同じ方法で受け渡しが行われる.

3.5 Access Control List の管理形態

CAS-ACL は CAS^2 の起動時に読み込まれるだけでなく, CAS^2 の動作を妨げることなく随時更新可能である. この機能を実現するために, 我々は CAS^2 に新規のサーブレット（Admin）を導入した. Admin サーブレットにアクセスすることにより, CAS^2 内部の CAS-ACL データベースの更新が可能となる.

また, Admin サーブレット自身も CAS-ACL による権限管理を受けるが, **`cas-auth-type: trusted`** で示される, 通常とは異なる CAS-ACL の制御に従う.

- **`trusted`** な CAS-ACL エントリ (1):

```
dn: ou=cas,o=NU
cas-allow: (uid=naito)
cas-auth-type: trusted
```

この CAS-ACL は LDAP DIT 内において CAS-ACL に関する subtree の root node であり, ここに示された **`uid=naito`** に一致するユーザは, CAS-ACL の全エントリをアップデートする権限を持つ.

さらに, Admin サーブレットは, CAS-ACL の分散管理も可能な仕様を持つ.

- **`trusted`** な CAS-ACL エントリ (2):

```
dn: ou=uPortal,ou=cas,o=NU
cas-allow: (uid=kajita)
cas-auth-type: trusted
```

上に示した CAS-ACL は, **`ou=uPortal,ou=cas,o=NU`** 以下の subtree の root node であり, ここに示された **`uid=kajita`** に一致するユーザは, この subtree 以下の全エントリをアップデートする権限を持つ. このように, Admin サーブレットによる CAS-ACL の管理機構は, CAS^2 を利用するアプリケーションの管理者たちによる分散管理を実現している.

4 名古屋大学ポータルでの実運用結果

ここでは, CAS^2 を用いた実運用結果として, 名古屋大学ポータルおよび学務情報システムを用いて行った, 2004 年度後期成績入力・2005 年度前期履修登録の実運用結果を報告する.

4.1 名古屋大学ポータルと学務情報システム

4.1.1 システム概要

名古屋大学ポータル ([2], 以後 MyNU と省略する.) とは, 名古屋大学において, 種々のウェブアプリケーションサービスの統合的な窓口を提供するために構築された, 「ポータルサイト」であり, 2004 年末から実験運用が始まり, 2005 年 2 月には学務情報システムの稼働とあわせて正式運用に移行した[19]. 現在 MyNU が提供しているシステムは学務情報シ

[19] MyNU は名古屋大学情報連携基盤センター内のポータル専門委員会によって運営されている. 筆者たち 2 名はポータル専門委員会の構成メンバーである.

ステムのみであるが, 今年度 (2005 年度) 内には学内の複数のシステムが MyNU に参加予定である.

一方, 学務情報システムは名古屋大学における履修・成績情報を統合的に運用するために計画されたシステムで, 各学部事務への履修・成績情報の提供のほか, 授業担当教員自身による成績入力, 学生自身による履修登録などをウェブアプリケーションとして提供するシステムである[20]. このシステムは, 2005 年度後期成績入力から実運用が行われた.

MyNU のようなポータルサイトでは, ユーザ認証を行った後には, ユーザに適応した情報を提供することが必要となる. 例えば, 学生がログインした場合には「履修登録や成績情報の照会などのサイト」へのリンクを表示し, 一方, 教員がログインした場合には「成績入力のサイト」へのリンクを表示するなど, ユーザ（またはユーザのクラス）によって異なる適切な表示を行うことが重要な役割となる. また, 学務情報システムでは明らかにユーザ認証が必要となる.

したがって MyNU と学務情報システムを関連させて運用する際には, それぞれがユーザ認証を必要とすることになるが, ここで, シングルサインオンを実現するために CAS^2 を利用した.

4.1.2 機器構成と CAS^2 の利用

MyNU と学務情報システムの機器構成は, 以下の表の通りである.

MyNU		
CAS^2 及び LDAP サーバ	Sun Fire V480	1 台
CAS^2 サーバ	Sun Fire V120	(Hot Standby 1 台)
LDAP サーバ	Sun Fire V120	(Hot Standby 1 台)
ウェブサーバ	Sun Fire V210	4 台
データベースサーバ	Sun Fire V210	1 台 (+ Hot Standby 1 台)
学務情報システム		
ウェブサーバ	Sun Fire V210, V120	各 2 台
データベースサーバ	Sun Fire V240	1 台

Table 1: 名古屋大学ポータル及び学務情報システムの機器構成

これらの機器を以下の図 (Figure 13) のように, Nortel Networks 社の Alteon によって負荷分散及びホットスタンバイを行った[21].

[20]学務情報システムは, 全学レベルの学務情報システム専門委員会および名古屋大学学務部学務企画課によって運営されている. 筆者たち 2 名は学務情報システム専門委員会の構成メンバーでもある.

[21]MyNU および学務情報システムへの外部からの通信は, 全て SSL を用いて行われる. Alteon は SSL アクセラレータとしても機能し, SSL を利用しない通信は, Alteon より内部に入ることはできない.

また, Alteon より内部のネットワークは, 完全な閉鎖ネットワークとなるため, MyNU および学務情報システム内部では SSL は利用しなかった.

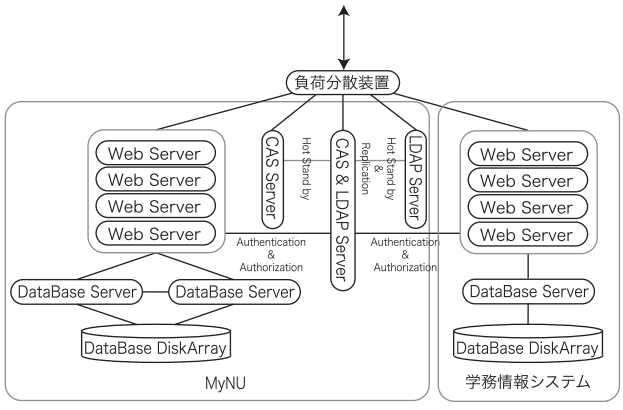

Figure 13: MyNU と学務情報システムのネットワーク構成

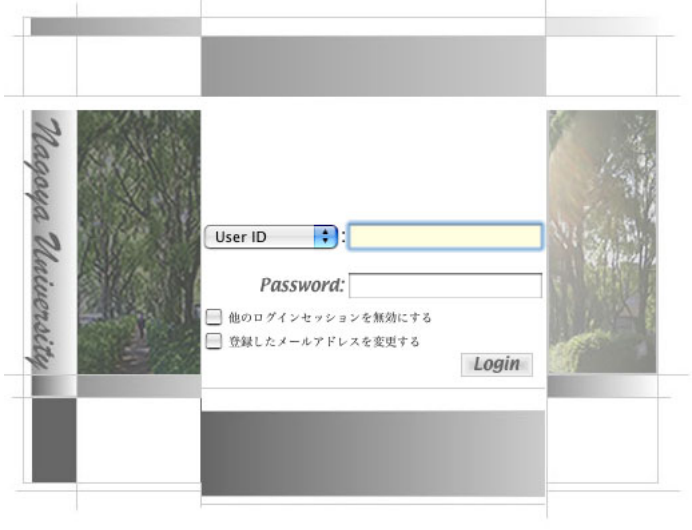

Figure 14: MyNU Login Window

また, MuNU はオープンソースの大学ポータルフレームワークである uPortal を用いて構築され, そのデータベースには Oracle 10g を利用し, データベースサーバとデータベースが格納されるディスクアレイとの間をファイバチャネルで結んだクラスタ構成を行った. これにより, 主データベースサーバのトラブル時には, ホットスタンバイ機器とのシームレスな連携が可能となった. uPortal は Java によるアプリケーションであるため, uPortal に Java CAS^2 クライアントを組み込み, MyNU での CAS^2 認証を行った.

学務情報システムのウェブアプリケーションは, Oracle(9i) のスクリプト言語である PL/SQL によって記述されているため, PL/SQL CAS^2 クライアントを組み込み, MyNU の CAS^2 サーバを利用した CAS^2 認証を行った.

4.2 実運用結果

名古屋大学における, 平成 16 年度後期成績入力は, 授業担当教員約 1000 名, 科目数約 4000 を対象に, 2005 年 2 月中旬に 19 日間 (464 時間) で行い, アクセス元を名古屋大学学内

からに限るという制限を設けて運用した. また, 平成 17 年度前期履修登録は, 新 2 年生から新 4 年生, 約 6500 名を対象に, 2005 年 3 月下旬に 9 日間 (203 時間) で行い, 学内だけでなく, 学外からのアクセスも許可して運用した. ともに, 各日 2 時間程度の保守時間を設定した以外には, 深夜のアクセスも許可した.

また, CAS^2 の「ログイン機能」として,「電子メールアドレス」をログイン ID とする認証機能を提供して運用を行った[22]. （3.4 節参照）

4.2.1 運用内容

成績入力及び履修登録における, ユーザのアクセス手順は以下の通りである.

成績入力

1. MyNU へのアクセス ⟶ CAS^2 認証 ⟶ MyNU 個人ページの表示 （ここに学務情報システムへのリンクが存在する）
2. 学務情報システム「教員メインメニュー」の表示
3. 「担当科目一覧」の表示
4. 履修者の成績入力 （1 ページあたり 25 名の表示を行い, ラジオボタンによる入力）⟶ 入力後, 担当科目一覧ページへ戻る

履修登録（通常科目）

1. MyNU へのアクセス ⟶ CAS^2 認証 ⟶ MyNU 個人ページの表示 （ここに学務情報システムへのリンクが存在する）
2. 学務情報システム「学生メインメニュー」の表示
3. 「通常科目（曜日・時限が決まった科目）入力ページ」の表示（時間割表の表示）
4. 曜日・時限の選択
5. 科目コード入力 ⟶ 科目名の表示と確認要求 ⟶ 登録入力後,「通常科目入力ページ」の表示

履修登録（集中講義科目）

1. MyNU へのアクセス ⟶ CAS^2 認証 ⟶ MyNU 個人ページの表示 （ここに学務情報システムへのリンクが存在する）
2. 学務情報システム「学生メインメニュー」の表示
3. 「集中講義入力ページ」の表示
4. 科目コード入力 ⟶ 科目名の表示と確認要求 ⟶ 登録入力後,「集中講義入力ページ」の表示

ここからわかるように,「成績入力」に関しては, ユーザ数が比較的少ないことと, 実際に行うアクションが少ないことから, システムに対する負荷は大きくないと判断できる.

[22]「ユーザ ID」,「電子メールアドレス」のいずれでもログイン ID として利用可能とした. どちらを利用するかは「ログイン画面」(Figure 14) のプルダウンメニューで選択可能とした. なお,「電子メールアドレス」は, MyNU 内で「電子メールアドレス登録」を行えるように設定し, そこで登録した任意の電子メールアドレス（この時点で LDAP サーバに登録）をログイン ID として利用できるようにした. なお, LDAP サーバ内の電子メールアドレスを検索する段階で, そのアドレスに一致するエントリが複数見つかった場合には, ログインは失敗するように設定してある.

一方,「履修登録（通常科目）」は, 1 名のユーザが行うアクション数が非常に多く, 科目コードから科目名・担当者名を表示する際にデータベーステーブルの検索が行われるため, システムに大きな負荷が発生する. また, ユーザ数が比較的多いばかりでなく, 履修登録締切り最終日にアクセスが集中することが予測される.

4.2.2 限界性能試験

このようなアクセス集中時のシステムの挙動を把握するため, 履修登録に先立ち, e-TEST suite (cf. [3]) を用いた負荷実験を行い, 次の表のような実験結果を得た. なお, 測定値としては, ボトルネックとなることがあらかじめわかったデータベースサーバの CPU 使用率が 85%以上に達した時の値を採用した.

処理内容	スループット（ページ／秒）	レスポンス（秒）
履修登録	37.5	1.5
集中登録	60.0	1.1
ポータル	17.5	2.4

Table 2: 負荷試験結果

また, CAS サーバのサーブレット呼び出し回数も同時に計測し, 集中登録時に 3000 回／分の呼び出しが行われていることを確認した.

この結果から, 以下に挙げるアクセスを正常に処理できることがわかった.

- 1 秒あたり 10 人以上の MyNU への新規ログイン.
- 150 人以上の同時にアクションを行うユーザ. (1 ページあたりの思考時間を 5 秒と考えた.)

この結果からわかるように, 外部認証システムである CAS[2] 認証を行ったとしても, 高負荷が予想される履修登録において十分な性能が得られることがわかった[23].

4.2.3 実運用時での CAS[2] の性能

成績入力・履修登録の実運用時の CAS サーバのサーブレット呼び出し回数は以下の表の通りであり, 限界性能試験時の 3000 回／分の実績と比較して大幅な余裕があったことがわかる.

運用内容	最大（回／分）	合計（回）
成績入力	500	158559
履修登録	1236	1418207

Table 3: 実運用時の CAS 呼び出し回数

[23] この結果に従い, 学務情報システムの Oracle 9i サーバのチューニングパラメータを設定した. 特に重要な「同時実行可能な SQL リクエスト数」の値を 150 に設定した.

また, 履修登録期間 (10 日間) の CAS[2] サーバへのアクセス数の時間平均は下図のようになった[24].

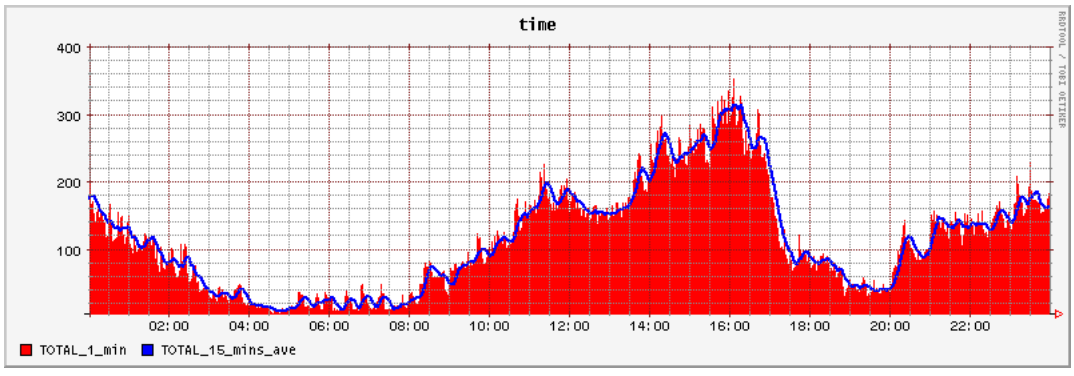

Figure 15: 履修登録期間のアクセス数（平均値）

また, 履修登録時における CAS サーバのサーブレット反応時間についても計測を行った結果, Login サーブレットについては, 平均 0.2 秒程度, Validation サーブレットについては, 平均 0.05 秒程度で推移し, アクセス数の増大に伴って急激に反応時間が低下するなどの現象は見られなかった.

4.2.4 実運用時に得られた結果

CAS[2] は詳細なアクセスログを残す機能を持つ. そのため, 実運用時にアクセス元のプロバイダやブラウザの種類など, 以下のデータを得ることができた[25].

アクセス元のプロバイダ		
1919	(25.0%)	名古屋大学情報メディア教育センター
982	(12.8%)	`net.bbtec`
667	(8.7%)	(不明)
622	(8.1%)	`jp.ne.dion`
597	(7.8%)	`jp.ne.ocn`
402	(5.2%)	`jp.ne.starcat`
235	(3.1%)	(学内の不明なホスト)
207	(2.7%)	`jp.ne.so-net`
197	(2.6%)	`jp.ne.aitai`
アクセスに利用されたブラウザ		
4199	(57.0%)	`Windows.XP.MSIE`
1757	(23.9%)	`Windows.2000.Netscape`
757	(10.3%)	`Windows.98.MSIE`
201	(2.7%)	`Windows.2000.MSIE`

[24]毎日 18 時から 20 時は「保守時間」に指定したため, この時間帯のアクセス数は比較的少なくなっている. 「保守時間」と指定したが, 実際にシステム保守を行うことはなく, この時間帯においてもシステムへのアクセスは可能であった.

[25]数値は「ユーザ, アクセス元のプロバイダ」および「ユーザ, ブラウザ名」をキーにした実データである. すなわち, 同一ユーザが同一プロバイダからアクセスした際には “1” としかカウントされていない. これらは, それぞれの上位数件のデータである.

Table 4: 実運用時に得られた結果

この結果からわかるように, 履修登録の全アクセスのうち, 学内からのアクセスは約 30%程度であり, 学外からのアクセスを許可したことによる利便性の向上が認められる.

5 まとめと今後の課題

本稿では, 多様な管理形態を持つ情報システムの統一的かつセキュアな認証基盤の実現の一つの方法として, 我々が拡張した CAS^2 を用いて構築した認証基盤について論じた. 我々が行った CAS への権限管理機構の組み込みは, Queens' University などでも検討されており, Central Authentication and Authorization Service の必要性の存在が明らかであると考えられる.

今回の CAS^2 拡張は, CAS Version 2 を基本として行ったが, 新たに発表された CAS Version 3 では, Spring Framework による記述に全面的に変更され, 種々の機能追加・動作方法の変更などが容易に実現できるようになった. したがって, 我々の CAS^2 拡張も Spring Framework 内で実現することにより, より容易に高機能な Central Authentication and Authorization Service の実現が可能であると考えられる.

また, 今回の CAS^2 拡張で見送った機能拡張の例として次のようなものが挙げられる.

- CAS サーバ自身の負荷分散.
  今回の実運用よりもさらに高負荷環境で CAS を利用するためには, CAS サーバ自身を負荷分散対象とする必要があるであろう. そのためには, TGC/ST を保持するデータベースを外部に持ち出すか, Java RMI などのリモート呼び出し機構などを用いて TGC/ST の共有を行うなどの方法が考えられる.
- 広域認証網での CAS の利用.
  大学においては, 他大学からの訪問者に対してサービスを提供しなければならない場面も少なからずあり得る. そのような場面においては, 他大学などで動作している CAS サーバを経由して認証結果などの交換が必要となる.

また, 今後の検討課題としては, 「認証強度」を複数設定することも考えるべきであろう. 現在の認証方法は, CAS サーバが認証するアプリケーション全てに対して同一レベルの認証でアクセスが可能である. しかしながら, 今後多様なアプリケーションに対して CAS 認証を利用することになると, 単なる「パスワード」だけでよいアプリケーションだけでなく, より強度の強い認証を利用したいアプリケーションも扱う必要がある[26]. このような必要な「認証強度」をアプリケーションごとに指定して CAS 認証を行うことも不可能ではないと考えられる.

これらウェブアプリケーションに対する多様で大規模・広範囲な統一認証基盤として, CAS は重要な役割を果たすと考えられる.

[26]「より強度の強い認証」の一例としては, 「SSL クライアント証明書」を利用した認証であったり, 「バイオメトリクス認証」である.

参考文献

[1] Yale University ITS Technology & Planning,
http://tp.its.yale.edu/tiki/tiki-index.php.

[2] 名古屋大学ポータル
https://mynu.jp/.

[3] e-TEST suite
http://www.fmw.fujitsu.com/services/etestsuite/.

[4] CAS Generic Handler,
http://esup-casgeneric.sourceforge.net/.

[5] JA-SIG, **http://www.ja-sig.org/**.

[6] Central Authentication Service,
http://jasigch.princeton.edu:9000/display/CAS.

[7] Internet2 Working Group, Shibboleth Architecture,
http://docs.internet2.edu/doclib/draft-internet2-mace-shibboleth-architecture-05.html.

[8] CAS Generic Handler,
http://esup-casgeneric.sourceforge.net/.

[9] 梶田 将司, 内藤 久資, 小尻 智子, 平野 靖, 間瀬 健二, CAS によるセキュアな全学認証基盤の構築, 情報処理学会研究報告, Vol. 2005, No. 39, pp. 35-40 (2005).

[10] 梶田 将司, 内藤 久資, 小尻 智子, 平野 靖, 間瀬 健二, CAS によるセキュアな全学認証基盤による名古屋大学ポータルの運用, 第 3 回 WebCT Conference 予稿集, pp. 115-120 (2005).