修士論文

パスワード埋込み画像による
ログインシステムの実装と評価

平成 17 年 2 月 21 日 受理

島根大学大学院　総合理工学研究科

数理・情報システム学専攻　計算機科学講座

指導教官　田中　章司郎

喜代吉　容大

目次

第１章　序論

現在、ユーザ毎に異なったサービスを提供する場が増え、ユーザ認証はなくてはならない存在になっている。ユーザ認証とは、あらかじめ本人であることを登録しておき、認証に必要な情報（ユーザ名やパスワードなど）を提示することで本人であるかを確認することである。認証方法には大きく３つの種類に分類できる（表 1.1）[1]。

表 1.1　ユーザ認証の種類

種類	例
パスワード認証	パスワード
カード認証	磁気カード・IC カード
バイオメトリクス認証	指紋・声紋・網膜パターン照合

パスワード認証およびカード認証では、盗用、忘失、破損、偽造といった危険がある。また、バイオメトリクス認証ではそれらの可能性は少ないが、ユーザの心理的圧迫感や、導入コストといった問題が考えられる。このように、認証技術を考えるには安全性だけでなく、簡便性、導入コスト、ユーザの心理的圧迫感なども考慮する必要がある。

1.1　研究概要

近年、ブロードバンドの普及や、インターネットの高速化が進み、多くの Web サイトを閲覧する機会が増えた。また、セキュリティを考慮するために、ID とパスワード（ログイン情報）によるログインシステムが少なくない。従って、インターネット上で情報を安全にやり取りする必要性が従来にも増して高まっている。また、安全な技術が確立すれば、今にも増して利便性の高い Web サービスが提供されると考えられる。

しかしながら、パスワード認証では、ユーザは多くのログイン情報を暗記する必要があり、パスワード認証を用いた Web サイトが増加するにつれて、ユーザのログイン情報の暗記・入力の負担も増加する。そのため、複数のサイトでの同じパスワードの使用や、PC に直接メモしているユーザも多いと考えられる。

そこで本研究では、パスワード認証をベースとし、安全性、簡便性、導入コスト、ユーザの心理的圧迫感なども考慮した、新しいログインシステムを提案する。これは、ステガノグラフィを用いてログイン情報を画像に隠蔽し、その画像（ログイン画像）を送信することで簡単にログインできるシステムである。ユーザはログイン情報を暗記する必要はない。また、画像であるため、第３者が画像を見て直接ログイン情報を得る危険はない。さらに基本的なパスワード認証をベースとしているため、ユーザの心理的圧迫感も少ない。画像の送信方法も一般的なブラウザの機能を使用しているため、導入コストも低い。

1.2　先行事例との比較

現在、すでに多くのユーザ認証技術が提案されている。そこで、INSPEC（英国電気学会 IEE が作成している電気・電子・制御・情報工学関連の英文二次文献データベース）で調査した（表 1.2）。

表 1.2　先行研究検索結果

キーワード	検索結果 (件)
(authentication)	8633
(authentication)and(login)	99
(user-authentication)	536
(user-authentication) and (login)	23
(steganography) and (login)	0
(steganography) and (authentication)	0

表 1.2 の中で、検索結果が 23 件のものに着目したところ、その内 9 件はカード認証およびバイオメトリクス認証に関するものであった。残りの文献について、ユーザ認証として強く関連のある文献を参考のため記載する（表 1.3）。

表 1.3　関連研究

SSH-secure login connections over the Internet, by Ylonen, T. (1996)
A remote user authentication scheme using hash functions, by Cheng-Chi Lee, Li-Hua Li, and Min-Shiang Hwang (2002)
Accessing MS-DOS applications over a TCP/IP network, by Ozimek, I. (1996)
Security considerations in the delivery of Web-based applications : a case study, by Shubert Foo, Peng Chor Leong, Siu Cheung Hui, and Shigong Liu (1999)
An enhancement of timestamp-based password authentication scheme, by Lei Fan, Jian-Hua Li, and Hong-Wen Zhu (2002)

さらに、「steganography」というキーワードをもとにさらに類似先行研究を調査したが、いずれもステガノグラフィを用いた画像によってログインする技術は見受けられなかった。ここで、本研究および先行研究調査にあたって、「ステガノグラフィ」というキーワードがしばしば出てくるが、これは画像にログイン情報を隠蔽する手法として画素置換型ステガノグラフィを用いたためである。従って、以後ステガノグラフィとは、画像にログイン情報を隠蔽する技術であり、大意として情報隠蔽技術（Information Hiding）と捕らえて欲しい。 このステガノグラフィはログイン画像のデザインをユーザの嗜好性に合わせることを可能にする。このことは、従来のパスワードに比べ暗記しやすく、多くのログイン情報

の管理を助けると考えられる。また、ユーザ独自の画像を用いることで、第３者に画像がログイン画像であることを認知されにくくなる。

実際に実用化されている類似先行事例としてはCookieを用いた自動ログインシステムが挙げられる。Cookie はユーザのコンピュータにブラウザを通して一時的にデータを書き込んで保存させる仕組みである(http://wp.netscape.com/newsref/std/cookie_spec.html)。このデータにログイン情報を保存し、次回のログイン情報の入力を省き、自動的にログインする技術である。多くのブラウザでサポートされているため、汎用性は高く、簡便で導入コストも低い。しかし、通常 Cookie はユーザの見えないデータであり、Cookie データを意図しない目的で使用される不安などユーザの心理的圧迫感は高い。また、ホストを認証するものであり、第３者でもログインすることが可能である。そのため、ノート PC の盗難、紛失などの際に非常に危険である。

そこで本研究では、ステガノグラフィを用いた画像に着目した。ステガノグラフィは、画像以外にも音声ファイルなどに隠蔽することが可能な技術である。その中でも、画像ファイルはデジタルカメラの普及や携帯電話のカメラ機能の向上により、画像ファイルが PC 上に存在することは自然である。従って、ログイン画像が PC のデスクトップ上にあることは、一見して普通の画像ファイルであるため、第３者から見ても自然であり、それがログイン情報であることは認知されにくい。また、音声ファイルと比較しても、容量が軽く、インターネットを通した送受信に適していると考えられ、後述するようにフラッシュメモリ等で容易に移動可能である。

また、同じようにログイン情報の暗記・入力の手間を省く Cookie との違いは、ログインが自動ではないことである。このことは、画像にログイン情報が隠蔽されていることを知る本人のみが利用できることを意味する。

第２章　ログイン画像

2.1　ステガノグラフィ

ステガノグラフィとは、画像や動画、音声などのマルチメディアデータに、画質や音質にはほとんど影響を与えずに特定の情報を埋め込む技術のことである。また、画像に著作権情報などを隠蔽する電子透かしという技術が存在する。これらは、使用目的が異なるだけで、技術的には同一のものである[2]。

また、静止画像ファイルにおけるステガノグラフィ技術には、大きく別けて以下の３つの方法がある[3][4][5]。

(1)画素置換型ステガノグラフィ
(2)周波数特性を利用したステガノグラフィ
(3)輝度情報を利用したステガノグラフィ

本研究では、(1)の画素置換型ステガノグラフィを用いて 24bit フルカラーBMP 画像（以下 BMP 画像）サイズ 32×32 にログイン情報を埋め込んだ。

2.2　隠蔽構造

BMP 画像において、１画素は RGB 値をそれぞれ８bit で構成されている。画素置換型ステガノグラフィは、その８bit 中の１bit を変更することでデータを隠蔽する技術である。本研究では埋め込む位置を、最も隠蔽されていることが認知されにくい最下位１bit 目とし、データを順に隠蔽した（図 2.1）。

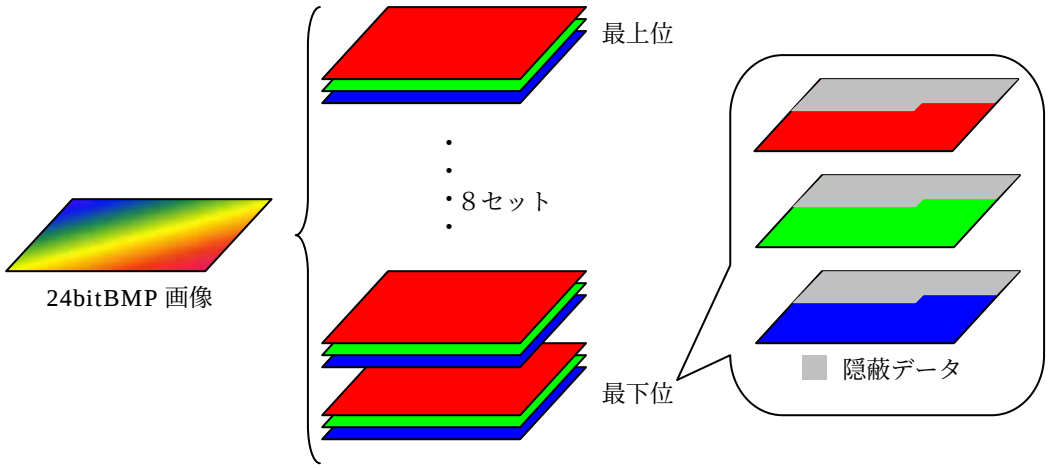


図 2.1　ログイン画像の隠蔽構造

2.3　ログイン画像

本研究で、ログイン画像に隠蔽されているデータを以下に示す（表 2.1）。

表 2.1　隠蔽データ内容

名称	説明
隠蔽キー	データが隠蔽されていることを示すキーワード データ抽出時に正当な抽出行為であることを示すために要求される
隠蔽データ長	隠蔽データの長さ
隠蔽データ	隠蔽データ（ID・パスワード）

本研究では、BMP 画像を構成する RGB 値を 8 bit 毎に順に取り出し、隠蔽キー・隠蔽データ長・隠蔽データの順に最下位 1bit 目を以下の法則に従って隠蔽している（表 2.2）。

表 2.2　隠蔽法則

元データ	隠蔽データ	隠蔽後のデータ
0	0	0
0	1	1
1	0	0
1	1	1

以上の方法でデータを隠蔽・抽出するプログラムを C 言語で作成した。以下それぞれ隠蔽プログラム「Suppression.exe」・抽出プログラム「Extraction.exe」と称す。また、参考までに通常画像とログイン画像を示すが、一見してログイン画像であることは分からない（図 2.2）。

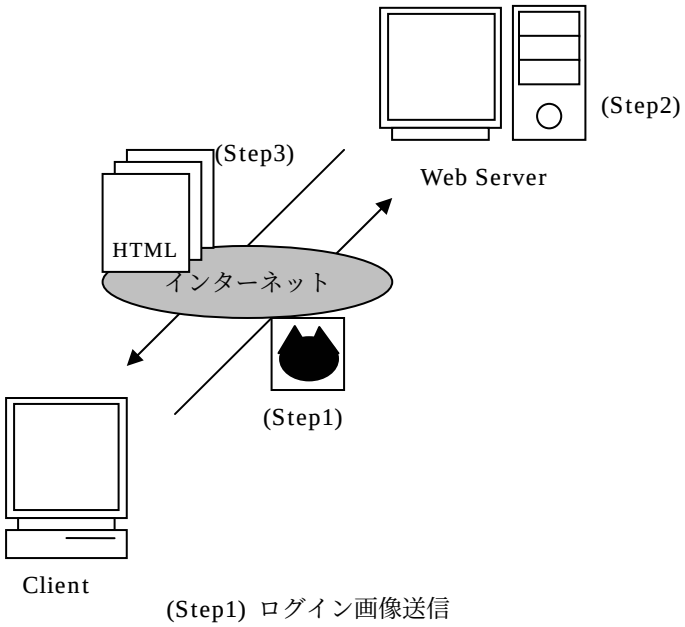
図 2.2　通常画像とログイン画像

第３章　実装方法

3.1　システム概要

本ログインシステムでは、ユーザはログイン画像をブラウザからサーバに送信することで簡単にログインできる（図 3.1）。また、ユーザのログイン情報暗記・入力の手間を省くため、ログイン情報はユーザに通知せず、ログイン画像のみを渡すことにした。このことから、ログイン画像は、外見的には一般的な画像であり、ユーザに心理的圧迫感を与えずに、アカウント情報を管理できる。

従来のパスワード認証と大きく異なる点として、外出先 PC からログインするには、ログイン画像は常に持ち歩く必要がある。パスワード認証では、ログイン情報を覚えておけば問題ない。この点については、近年フラッシュメモリなど、安価で大容量のデータを用意に持ち運ぶことが可能になり、さらにそれ自体にセキュリティ対策が施されているものもあり、ログイン画像の移動は容易であると考える。



(Step1) ログイン画像送信

(Step1) ユーザ認証

(Step3) 動的な認証結果の受信

図 3.1　ログインシステム概要

3.2 サーバシステム構成

本研究におけるログインシステムは、「新規アカウント作成処理」と「ログイン処理」の２項目で構成される。それぞれの処理内容は以下のとおりである（表 3.1）。

表 3.1　サーバ処理一覧

処理	内容
新規アカウント作成処理	新規アカウント作成からクライアントにログイン画像をダウンロードさせるまでの一連の処理
ログイン処理	ログイン画像受信からユーザ認証結果をクライアントに返すまでの一連の処理

また、ログインシステムを作成するにあたって、Java Servlet（以下 Servlet）を用いた。Servlet とは、Web サーバ上で実行される Java プログラムである。類似するものとして Common Gateway Interface（CGI）がある。どちらもサーバ上で実行され、動的に結果をクライアントに返す。しかし、CGI は呼び出された数だけプロセスが立つのに対し、Servlet は１つのプロセス内で実行する。従って、Servlet はサーバに常駐するが、アクセスが集中したときに CGI に比べ負荷が少ないという利点がある。本研究では Servlet Server として、Jakarta Project の Tomcat を使用した(http://jakarta.apache.org/tomcat/)。

Tomcat は Servlet Server であるが、同時に Web Server の機能も兼ね備えている。一般的な Web Server は 80 番ポートを使用するのに対し、Tomcat では 8080 番ポートを使用する（変更可）。しかしながら、Tomcat の Web Server 機能は簡易的なものであり、通常は Apache や Internet Information Server（IIS）と連携させて使用する。今回は最も普及率の高い Apache を使用した。Apache と Tomcat の連携には、Apache Jakarta Tomcat Connector を使用した。この連携は、静的コンテンツを Apache に処理させ、動的コンテンツのみを Tomcat に処理させるものである。

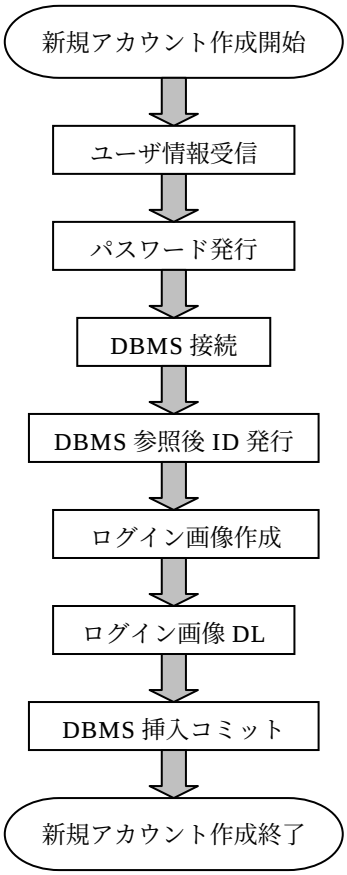
アカウント情報を管理する DBMS として、HITACHI HiRDB を用いた。また、Servlet からの DBMS アクセスとして、Java Data Base Connectivity（JDBC）を用いて接続した。以下にこれまでのサーバ構成を示す（表 3.2）。

表 3.2　システム開発環境

OS	Windows 2000 Server SP4
Web Server	Apache 2
Java Servlet Server	Tomcat 4
DBMS	HiRDB 7

3.3　新規アカウント作成処理

新規アカウント作成処理は、ブラウザを通して行われる。ユーザはブラウザの指示に従って必要事項を記入して送信することでアカウントは作成され、ログイン画像を入手することが可能である。記入事項は、ログイン後のサービス提供内容によって多少異なると考えられるが、ここでは、ユーザ名、キーワード、画像選択の３項目とする。キーワードは画像紛失時の対処として記入を要求した。新規アカウント作成処理のフローチャートを以下に示す（図 3.2）。



※　この新規アカウント作成における一連の処理において、どの時点で障害が発生しても、アカウントは正常に作成しないため、エラーとしてクライアントに返す。

図 3.2　新規アカウント作成処理のフローチャート

3.3.1　画像選択

画像選択とは、あらかじめサーバがユーザにログイン画像のデザインを複数提供し、ユーザが好みの画像を選択するという仕組みである。もちろんユーザによって好みがあるためであるが、それとは別に大きな意味がある。それはすべてのユーザが同じ Web サイトで同じログイン画像を使用した場合、第３者にその画像がログイン画像であることが一目で判ってしまうからである。サーバが提供するサンプル画像を増やすことで回避はできるが、それでも限界はある。また、サーバの画像サンプル提示で注意すべき点は、デザインを示すことは重要であるが、埋め込む元画像がユーザに漏洩してはならない。作成されたログ

イン画像は一目見てもログイン情報を抽出することは難しい。しかし、埋め込む元画像と比較し、差分をとることで、不正にログイン情報を抽出される危険がある[6]。今回はログイン画像が BMP 画像であることから、サンプル画像は JPEG に変換したものを使用することで対処した。以下に実際の入力画面を示す（図 3.3）。

図 3.3　新規アカウント作成入力画面

最終的には、サーバがログイン画像サンプルを提示するのではなく、ユーザの嗜好性を考慮するため、ユーザが独自で有する好みの画像を元にログイン画像を作成する必要がある。これは従来のパスワードと比較しても、画像とユーザにユーザだけが知る固有の関係性を持たせることができ、ログイン情報の管理が容易になると考えられる。しかし、ログイン情報を埋め込む元画像はユーザが有しているため、ログイン画像との差分をとられ、不正にログイン情報を抽出される危険がある。この対策については後に述べる。

3.3.2　ログイン画像作成

新規アカウント作成において、先に述べた隠蔽プログラム「Suppression.exe」を用いてログイン画像を作成する。クライアントの要求で Servlet から隠蔽プログラムを実行するにあたって、Java.Runtime.exec()関数を用いた。これは、Java プログラムから外部プロセスを実行し、結果を得る関数である。実行するプログラムはコマンドラインから実行でき、標準入力を必要とせず、実行後に必ず終了するものでなくてはならない。以下に実行例を示す（表 3.3）。

表 3.3　Java.Runtime.Exec()関数実行例

```
//実行コマンドを String 型に用意
String com = new String(“Suppression.exe D:¥Image¥Sample.bmp
                                D:¥Image¥Login.bmp 39395963tEkiToU7”);
//Runtime オブジェクト取得
Runtime rt = Runtime.getRuntime();
try {
   //コマンド実行後、終了まで待機
   Process pr = rt.exec(com);
   rt.waitFor();
}
//エラー処理
catch (Exception e) {
   System.exit(1);
}
```

また、Java.Runtime によってファイルを生成するため、ファイル書き込み許可をする必要がある。これは Tomcat のセキュリティ設定ファイル「Catalina.policy」に「Java.Runtime による書き込みを許可する」という内容を書き込めば良い。以下に設定例を示す（表 3.4）。

表 3.4　Catalina.policy 設定例

```
grant codeBase "file:${catalina.home}/LGW/*" {
    permission java.lang.RuntimePermission "java.lang.Runtime.*", "write";
    permission java.io.FilePermission "java.io.*", "write";
};
```

3.3.3　トランザクションの衝突

新規アカウント作成では、ID とパスワードを発行する。パスワードについてはランダムに発行している。また、ID においては８桁の整数を昇順に発行している。そのため、ID を昇順に発行する前に、DBMS に接続する必要がある。このとき、同時に複数の新規アカウント作成トランザクションが立ったとき、１つのトランザクションが DBMS 挿入コミットする前に他のトランザクションが ID を発行する可能性がある。このことから、同じ ID を持つアカウントが複数生成される危険がある。これを避けるために、新規アカウント作成では、Synchronized 宣言によって Servlet へのアクセスを制限した。以下にトランザクション衝突の例を示す（図 3.4）。

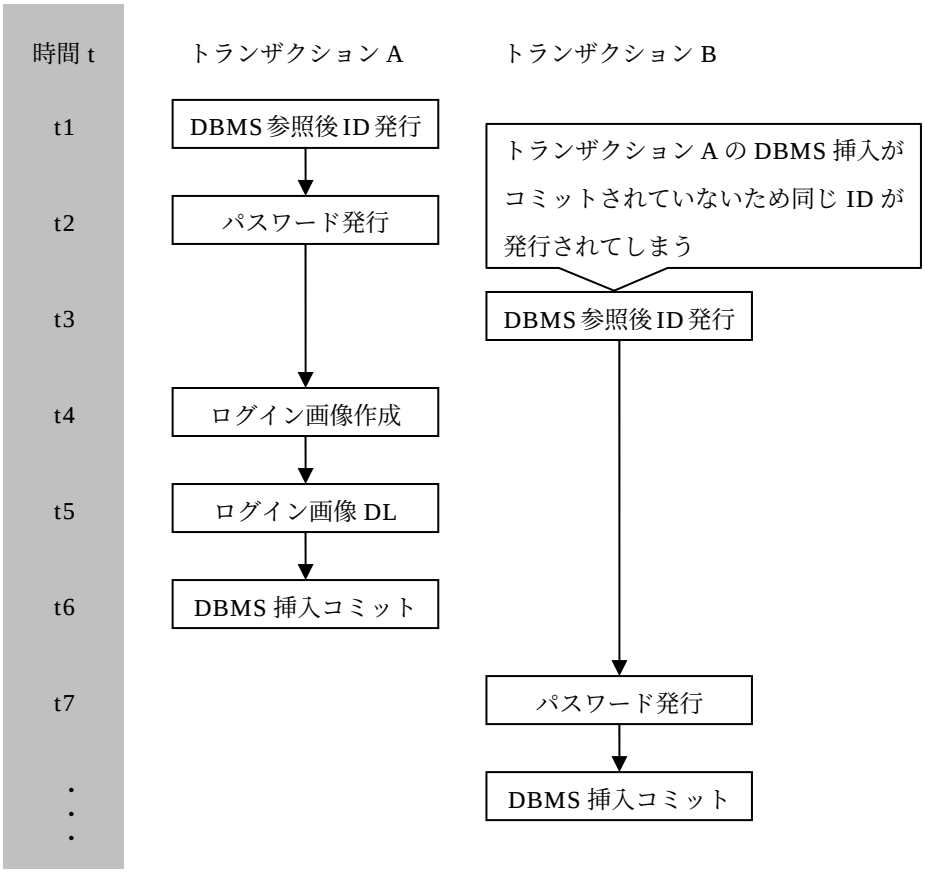


図 3.4　トランザクションの衝突

3.4　ログイン処理

ログイン処理も新規アカウント作成処理と同様にブラウザを通して行われる。HTML のファイル送信フォームを用いてログイン画像をサーバに送信するだけでよい（図 3.5）。

図 3.5　ログイン画面

なお、ログイン画像送信は HTTP メソッドの POST を使用した（RFC 2616）。以下にログイン処理のフローチャートを示す（図 3.6）。

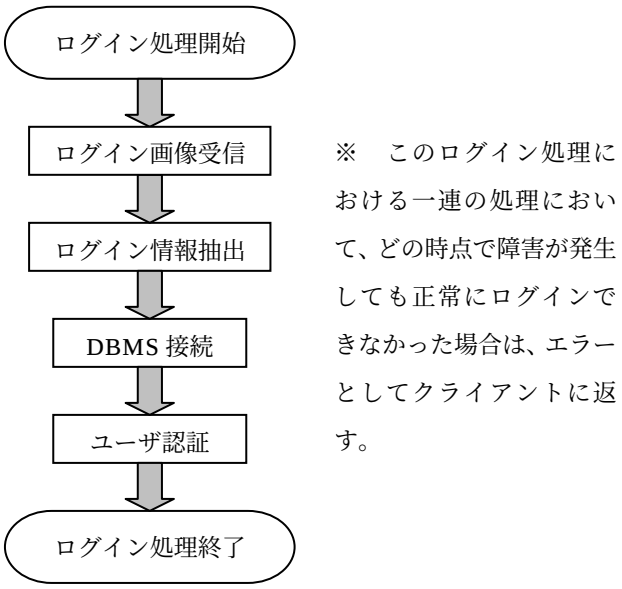


図 3.6　ログイン処理のフローチャート

3.4.1　ユーザ認証

ログイン処理におけるログイン情報抽出は、新規アカウント作成と同様の方法で抽出プログラム「Extraction.exe」を外部プロセスとして実行する。抽出したログイン情報は、DBMS に登録されているアカウント情報と比較し、ID とパスワードが両方一致した場合のみログイン成功クライアントに返す。ログイン処理は新規アカウント作成と異なり、あらかじめ DBMS 登録されているデータの参照のみ行う。そのため、他のトランザクションに影響を与えることはない。

参考のため、今回 DBMS に作成した Table を以下に示す（図 3.7）。

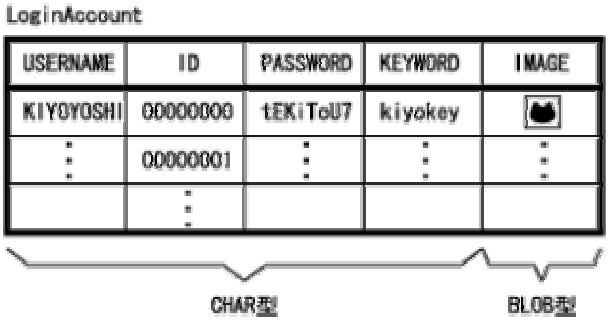


図 3.7　表定義

3.5　Java Data Base Connectivity

先に述べたように、Servlet から DBMS への接続は JDBC を用いている。JDBC は、Java プログラムからリレーショナルデータベースにアクセスするための API である。データベースの種類によらない汎用性の高いプログラムを開発することが可能だが、データベースに対応した JDBC ドライバが必要である。今回は、使用した DBMS である「HiRDB」付属の HiRDB JDBC Driver を使用した。

3.5.1　Driver Manager と Data Source

先に述べたように、本研究で Servlet から DBMS にアクセスするために JDBC を用いた。Servlet で JDBC を使用する方法として、以下の２つが存在する（表 3.5）。

表 3.5　Servlet における JDBC の使用方法

名称	説明
Driver Manager	個々の Java プログラムで JDBC を用いて、DBMS に接続する方法
Data Source	Data Source を用いて DBMS に接続する方法

どちらの方法も同じように、Connection オブジェクトを取得し、DBMS とやり取りを行うが、JDBC ドライバおよび、DBMS に接続するためのアカウント情報（本ログインシステムとは異なるユーザ名とパスワード）を定義する場所が異なる。

Driver Manager では、個々の Java プログラム内に JDBC ドライバを定義する。従って、接続は個々の Servlet（Java オブジェクト）が受け持ち、Servlet が生成されるたびに、DBMS への接続から切断までの処理が行われる。一方 Data Source では、Servlet Server である Tomcat に JDBC ドライバを定義する。従って、接続は Tomcat が一括して行い、Servlet は DBMS への接続から切断までの処理を行う必要はなく、１度接続したらその接続を保持し、他の Servlet に使わせることが可能である。さらに、コネクションプールや分散トランザクションを実装できる。また、DBMS や JDBC ドライバが変更された場合でも、アプリケーションのコードを変更する必要がないという利点もある。以下に、Driver Manager と Data Source の違いを示す（図 3.8）。

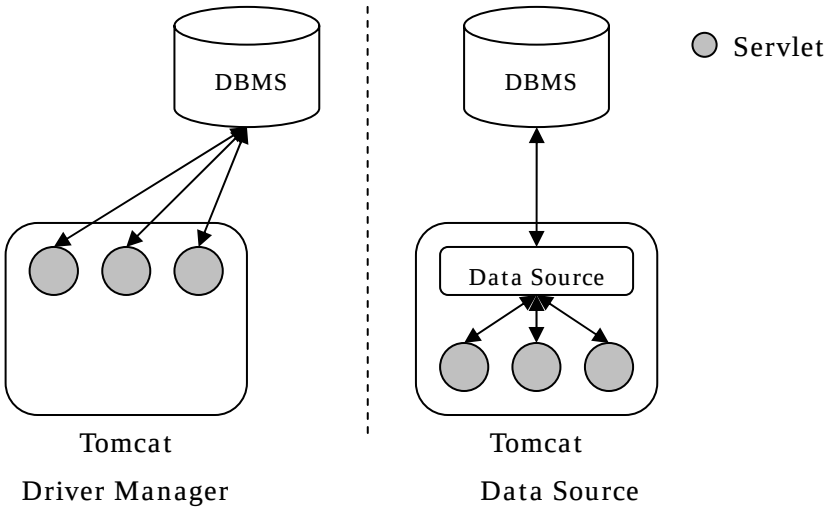


図 3.8　Driver Manager と Data Source の違い

Data Source を使用するためには Java Naming and Directory Interface（JNDI）が必要である。JNDI は Naming Service や Directory Service などのツリー構造の情報に統一的にアクセスするための使用である。以下に Tomcat で Data Source を用いた HiRDB における JDBC アクセスに必要な最低限の設定を示す（表 3.6～表 3.8）。

表 3.6　Web.xml の設定

```
<web-app>
  //デフォルトサーブレットマッピング
  <servlet-mapping>
    <servlet-name>invoker</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
  </servlet-mapping>

  //JDBC Data Source 設定
  <resource-ref>
    <res-ref-name>jdbc/datasource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</web-app>
```

表 3.7　Server.xml の設定

```
//アプリケーションフォルダ設定
<Context path="/LGS" docBase="LGS" debug="0"
                    reloadable="true" crossContext="true" trusted="false">
          <Logger className="org.apache.catalina.logger.FileLogger"
                    prefix="localhost_LGS_log." suffix=".txt"
                    timestamp="true"/>

//JDBC ドライバ設定
<Resource name="jdbc/datasource" auth="Container" type="javax.sql.DataSource"/>
  <ResourceParams name="jdbc/datasource">
    //DBMS ログインユーザ名
    <parameter>
      <name>username</name>
      <value>"root"</value>
    </parameter>
    //DBMS ログインパスワード
    <parameter>
      <name>password</name>
      <value>"root"</value>
    </parameter>
    //JDBC ドライバ名
    <parameter>
      <name>driverClassName</name>
      <value>JP.co.Hitachi.soft.HiRDB_Driver_for_JDBC.JdbcHirdbDriver</value>
    </parameter>
    //JDBC ドライバ URL
    <parameter>
      <name>url</name>
      <value>jdbc:hitachi:hirdb://localhost/:DSN=HRDB</value>
    </parameter>
  </ResourceParams>
</Context>
```

表 3.8　Java プログラムソース構造

```
import javax.naming.*;
import javax.sql.*;

public class Example extends HttpServlet {
  DataSource ds;
  // 初期化
  public void init() throws ServletException {
    try {
      // 初期コンテキストを取得
      InitialContext ic = new InitialContext();
      // ルックアップしてデータソースを取得
      ds = (DataSource)ic.lookup("java:comp/env/jdbc/datasource");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
    :
    :
```

3.5.2　トランザクション処理

本システムを構成する「新規アカウント作成処理」「ログイン処理」は、一連の処理がすべて正常に完了して初めて成功と言える。従って、途中で何らかのエラーが起きてしまった場合、それまでの処理や変更はコミットされてはいけない。本節では、JDBC を用いたトランザクション処理について述べる。

本システムでは、DBMS への問い合わせ言語として SQL を使用している。しかし、SQL は通常、１命令文を実行した時点で DBMS への変更は自動的にコミットされる。そこで、JDBC プログラミングでは、この自動コミット機能を外し、手動でコミットすることでトランザクション処理を実装する。また、コミットされるまでに起きたエラーはすべて Java プログラム内で処理する。以下に JDBC を用いたトランザクション処理の流れを示す（表 3.9）。

表 3.9　JDBC を用いたトランザクション処理の流れ

```
//DBMS 接続
Connection con = null;
Statement stmt = null;
try {
   // データソースから Connection を取得
   con = ds.getConnection();
   //自動コミット機能解除
   con.setAutoCommit(false);
   // Statement を取得
   stmt = con.createStatement();

   SQL 文実行処理を記述

   //手動コミット
   con.commit();
}
catch (Exception e) {
   e.printStackTrace();
}
finally {
   try {
      if (stmt!=null) { stmt.close();}
      if (con!=null) {con.close();}
   }
   catch (Exception e) {
      e.printStackTrace();
   }
}
```

以上　Tomcat に関する参考 URL

http://jakarta.apache.org/tomcat/

第４章　安全性

4.1　ログイン画像の安全性について

本システムでは、ログイン画像の送信によってログインできる。逆に、従来のパスワード認証のようにIDやパスワードといったログイン情報を入力するだけではログインできない。このことから、ログイン情報自体が漏洩することは危険ではないと考えられるかもしれない。しかし、ログイン画像から不正にログイン情報が抽出された場合、その抽出者はログイン画像の隠蔽構造を知っていると考えられる。つまり、ログイン画像を偽造することが可能であると考えられる。

また、ログイン情報の漏洩に関係なく、ログイン画像とログイン画像の送信先が第３者に漏洩した場合、第３者は正当な方法でログインすることが可能である。従って、パケットを盗聴される危険を考慮し、ログイン画像の通信は暗号化して行うことが前提である。また、万が一暗号が破られた場合でもログイン情報自体が漏洩しないことは、サーバシステムのトラブル時等のメンテナンスに有用であると考えられる。さらに、ログイン画像から不正にログイン情報を抽出されないことは、先に述べたように、ユーザの嗜好性に合わせた画像をログイン画像として使用することを可能にする。ここからは、以下の３項目に別けてログイン情報の漏洩経路と対策について述べる。

4.2　ログイン画像漏洩経路と対策

4.2.1　サーバからの漏洩

サーバから漏洩する危険のあるものは以下の２つである。

（１）埋め込む元画像
（２）抽出プログラム

（１）埋め込む元画像が漏洩した場合、第３者はログイン画像と埋め込む元画像の差分をとることでログイン情報を不正に抽出することが可能となる。同様に、（２）抽出プログラムが漏洩した場合、第３者はログイン画像に抽出プログラムを適用することでログイン情報を不正に抽出することが可能となる（図 4.1)。しかし、これら２つはサーバ上で管理され、漏洩しないことを前提とするため、ここでは触れない。

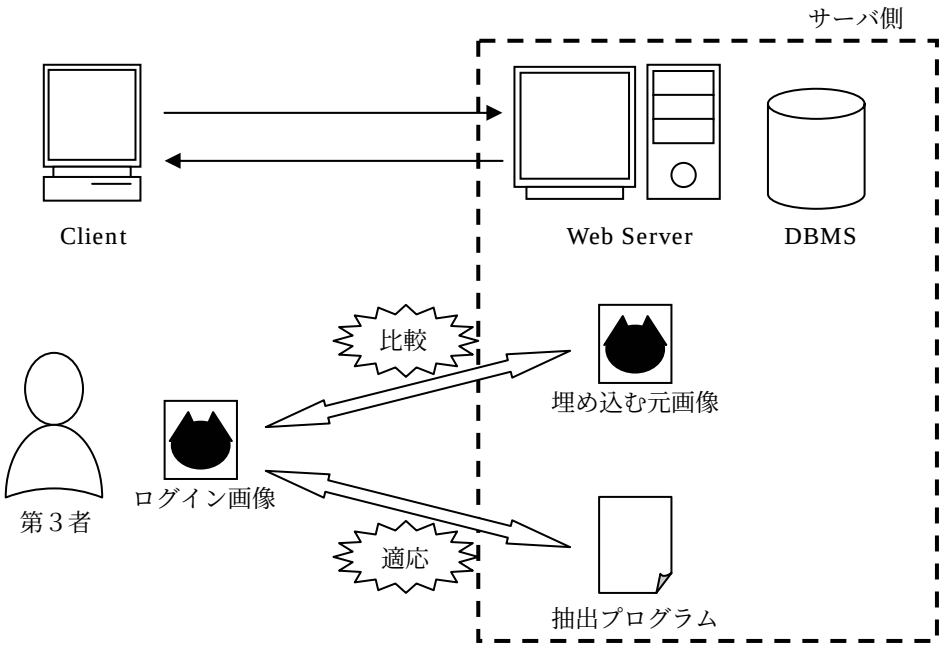


図 4.1　サーバからの漏洩

4.2.2　クライアントからの漏洩

クライアントから漏洩する危険性として挙げられるものは以下の２つである。

（１）１つのログイン画像の解析
（２）複数のログイン画像同士の比較

本システムでは、隠蔽データは画素置換型ステガノグラフィを用いて BMP 画像の最下位１bit 目に先頭から順に隠蔽されている。このことから、ログイン画像を解析し、RGB 値の最下位１bit を先頭から順に取り出すことでログイン情報は抽出される。従って、（１）１つのログイン画像の解析によって不正にログイン情報を抽出される危険がある。また、ログイン画像はすべて同じ法則で隠蔽されているため、（２）複数のログイン画像同士を比較することでログイン情報を不正に抽出される危険がある（図 4.2）

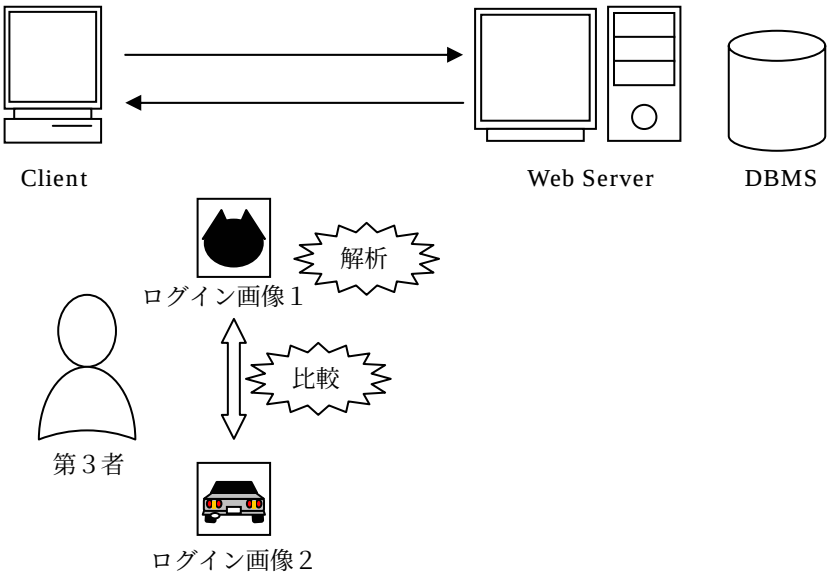


図 4.2 クライアントからの漏洩

（１）１つのログイン画像の解析に対する対策として、埋め込み位置を拡散することが考えられる。ここで、画素置換型ステガノグラフィによる埋め込みでの最大隠蔽容量について考える。

最大隠蔽容量 (bit) ＝（縦幅）×（横幅）×３

上記の式から、画像サイズを 32×32 としたとき、最大隠蔽容量は 3072bit となる。このことから、埋め込み位置を拡散した場合の埋め込む組み合わせは 3072!通りとなる。従って、１つのログイン画像の解析によって不正にログイン情報を抽出することは通常の画像の大

きさでは計算量的に困難となる（図 4.3）。

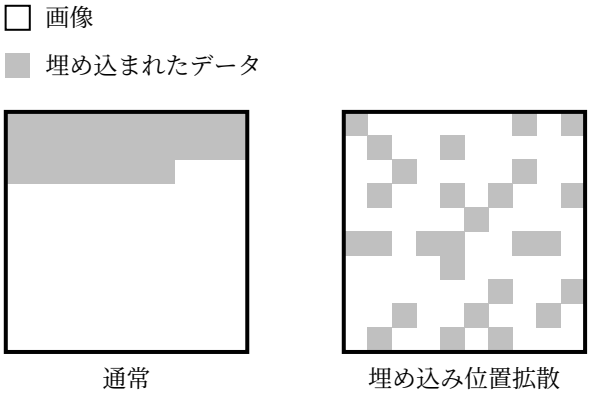


図 4.3　埋め込み位置拡散

次に（2）複数のログイン画像同士の比較について考える。現在の隠蔽方法はもちろん、先に述べたように埋め込み位置の拡散を行っているログイン画像でも、複数のログイン画像同士を比較することで、不正にログイン情報を抽出される危険がある。つまり、すべてのログイン画像が同じ法則で隠蔽されている限り、複数のログイン画像同士を比較することで、不正にログイン情報を抽出される危険がある。

対策としては、埋め込み位置を応用し、個々のログイン画像で埋め込む位置を変化させることが考えられる。埋め込む位置の組み合わせは先に述べた 3072!通りであり、複数のログイン画像同士を比較しても隠蔽の法則性を見つけることは計算量的に困難である。しかしこの方法では、抽出時に必要となる拡散情報をどうするかという問題が考えられる。

ここで、拡散情報は個々のログイン画像に対して一意である必要がある。一例として、ログイン画像作成時のタイムスタンプ情報を元に拡散位置を決める方法が考えられる。しかし拡散位置情報は位置情報の漏洩を防ぐために暗号化して画像に隠蔽する必要があり、かつ一意に復号化できる必要がある。この復号アルゴリズムがすべてのログインサーバに共通したものである以上、ログイン画像の埋め込み位置拡散による頑健性向上は認められない。従って、復号アルゴリズムがサーバ毎に異なる暗号化方法が求められる。しかし、本研究ではログイン画像の頑健性向上の可能性を埋め込み位置拡散によって見出したに留まり、ここでは拡散情報の暗号化についてこれ以上触れない。

さらに、（1）（2）に共通する対策として、ログイン情報を埋め込む前にあらかじめ暗号化してから埋め込む方法も考えられる。この方法では、埋め込む元画像との比較で隠蔽データを不正に抽出されても、ログイン情報は暗号化されているため第 3 者に漏洩することはない。また、埋め込む元画像との差分を取られても、第 3 者にログイン情報そのものは漏洩しない。従って、ユーザが独自に有する画像を使用することが可能となる。

4.2.3 通信中の漏洩

先に述べたように、ログイン情報の漏洩に関係なく、ログイン画像とログイン画像の送信先が第３者に漏洩した場合、第３者は正当な方法でログインすることが可能である。従って、パケットを盗聴される危険を考慮し、ログイン画像の通信は暗号化して行うことが前提である。

4.3 RSA 暗号との比較

本節では、公開鍵暗号で最も広く使われている RSA 暗号と比較することで、ログイン画像の頑健性を評価する。ここで頑健性とは、ログイン画像から不正にデータを抽出する攻撃を想定したとき、その行為が困難であると定義する。

RSA 暗号とは、巨大な数の素因数分解が計算量的に困難であることを利用した暗号である[7]。一般的に n は、10 進数で 310 桁の数(1024bit)とすると安全であるとされる。また、現在この暗号を解読するためのもっとも効率の良いとされるアルゴリズムは数体ふるい法による素因数分解であり計算量は $O(\exp(1.9(\ln(n))^{1/3}(\ln(\ln(n)))^{2/3}))$ である(http://www.rsasecurity.com/)。

一方、埋め込み位置を拡散したログイン画像の場合、不正な抽出にかかる計算量は最大隠蔽容量の階乗となる（4.2 節）。

ここで、RSA 暗号において n を 10 進数で 310 桁の数としたとき、ログイン画像においてこれと同等の計算量的安全性を得るには、最大隠蔽容量≥26 であれば良い（図 4.4）。

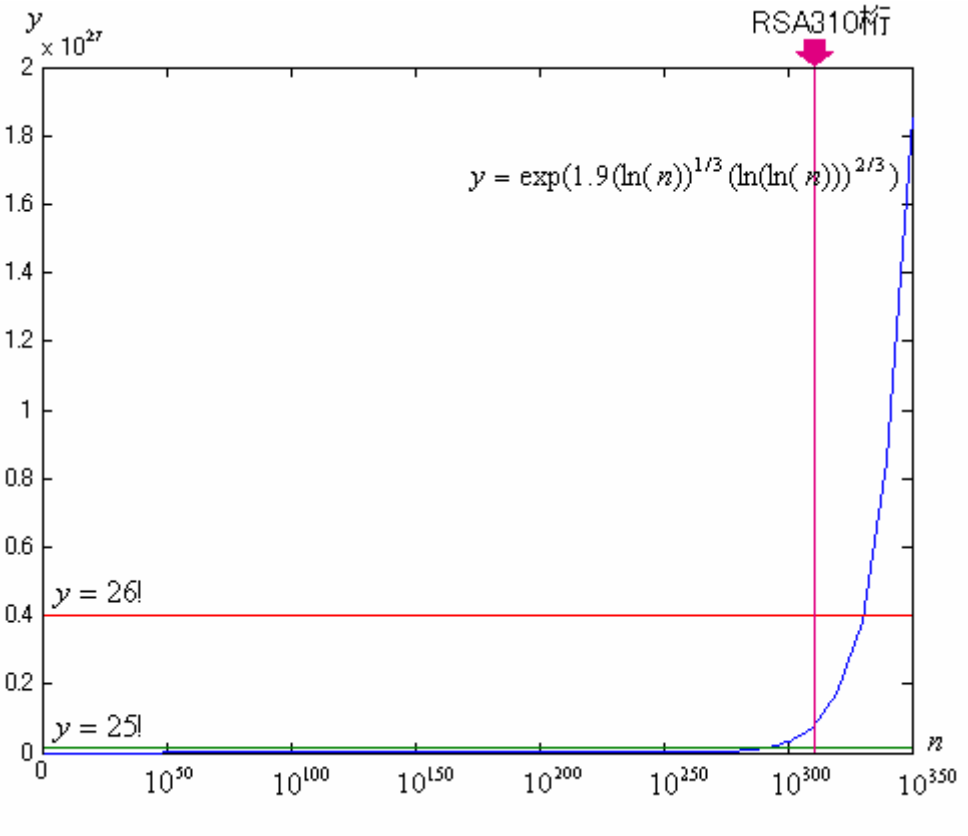


図 4.4　計算量のシミュレーション

従って、通常の大きさの画像を用いたログイン画像の場合、現在広く使われている RSA 暗号と比較しても、十分な頑健性を備えていると言える。また RSA 暗号と異なり、画像サイズを大きくすることで埋め込み位置の組み合わせも増加し、安全性も増すと考えられる。さらに、RSA 暗号を用いてあらかじめ暗号化したデータをログイン画像に隠蔽した場合、ログイン画像の頑健性は一層高くなる。

第５章　実験

5.1　実験内容

ログインサーバにクライアントからの擬似的アクセスを行い、その応答時間をクライアント側で測定することでサーバの耐久度を調べ、システムの実用性を検証する。ログインサーバに送信するリクエストは以下の３項目である（表 5.1）。

表 5.1　リクエスト項目

リクエスト名	内容
通常閲覧処理	Servlet を使用しない静的コンテンツの閲覧
ログイン処理	ログイン画像を送信してログインする処理
新規アカウント作成処理	新規アカウント作成してログイン画像を入手する処理

通常閲覧処理は、Java Servlet を使用しないログイン前の index.html への閲覧とした。通常閲覧処理の耐久度を測定することで､HTTP サーバのみの耐久度を調べることができる。また、Java Servlet を用いる動的コンテンツとの比較対象となる。

また、「ログイン処理」「新規アカウント作成処理」について、サーバ側で詳細測定を行い、処理時間のボトルネックとなる箇所を調査する。さらに、リクエスト成功率を調査し、これらの結果からサーバチューニングに役立てる。

5.2　擬似ブラウザ

擬似的アクセスによる実験を行うに当たって、「通常閲覧処理」「ログイン処理」「新規アカウント作成処理」を行う擬似ブラウザを作成した。擬似ブラウザは HTTP メソッドを用いてリクエストをサーバに送信する。さらに、マルチスレッドで実行することによって同時複数アクセスを可能にする。擬似ブラウザの作成には以下の３つの方法が考えられた。

（１）Winsock
（２）WinInet.dll
（３）Common Http Client

今回はプログラミングの容易さを重視し、Common Http Client(http://jakarta.apache.org/commons/httpclient/)を用いて擬似ブラウザを作成した。ログイン処理においてログイン画像送信する処理があるが、Http Client は文字だけでなく画像のようなバイナリファイルの送信をサポートしている。実際のプログラミングにおいては、FilePart というオブジェクトの生成によって行われる。以下に簡単な流れを示す（表 5.2）。

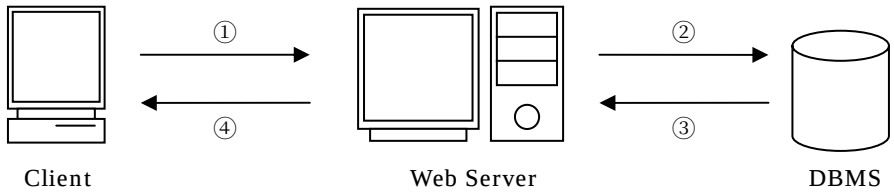
表 5.2　Http Client によるファイル送信

```
//Http Client 関連のインポートファイル
import org.apache.commons.httpclient.HttpClient;
import org.apache.commons.httpclient.HttpStatus;
import org.apache.commons.httpclient.methods.MultipartPostMethod;
import org.apache.commons.httpclient.params.HttpMethodParams;
import org.apache.commons.httpclient.methods.multipart.*;
…
//送信先アドレス・送信ファイル準備後に接続確立
String targetURL = new String("http://shinano/LGS/servlet/UserCheck");
File targetFile = new File("D:/ok.bmp");
MultipartPostMethod filePost = new MultipartPostMethod(targetURL);

try {
//FilePart オブジェクト生成後に POST メソッド実行準備
   FilePart part = new FilePart("filename", targetFile, "image/bmp", "");
   filePost.addPart(part);
   HttpClient client = new HttpClient();
   //ファイル送信実行
   int status = client.executeMethod(filePost);

   if (status == HttpStatus.SC_OK) { ファイル送信成功時の処理 }
   else { ファイル送信失敗時の処理 }
}
catch (Exception ex) { 例外処理 }
finally {   filePost.releaseConnection();}
```

ここでは特に示さないが、Http Client は、POST メソッドおよび GET メソッドによるテキスト送信もサポートしている。また、これらは Http Client を使用せず、通常の Java プログラムにおいて Java.net をインポートすることで容易に実装できる。しかし今回は、新規アカウント作成処理で GET メソッドを使用しており、計測環境の統一を目的として Http Client によって実装した。以下に擬似ブラウザによる計測ポイントを以下に示す（図 5.1）。



計測時間　＝　①＋②＋③＋④

図 5.1　擬似ブラウザによる計測ポイント

5.3　実験結果と考察

以下に実験環境を示す（表 5.3）。

表 5.3　実験環境

Server	
OS	Windows 2000 Server
CPU	PentiumⅡ 333MHz
RAM	512 M
Client	
OS	Windows XP Pro
CPU	Celeron 2.8GHz
RAM	512M
LAN	100 BASE-T

実験は 10 回行い、計測結果の平均値をとっている。また、初回起動時のコールドスタートはロードに時間がかかり、平均値に影響がでるため結果には含めていない。初回起動時の計測時間を参考までに以下に示す（表 5.4）。

表 5.4　コールドスタート時の計測時間

処理	計測時間(msec)
ログイン処理	2516
新規アカウント作成	3407

5.3.1　通常閲覧処理

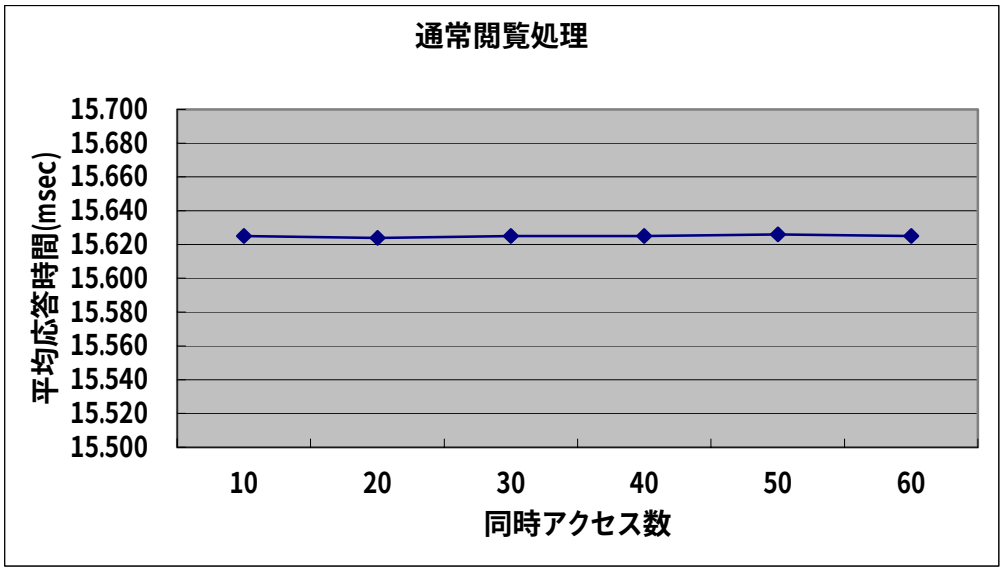


図 5.2　通常閲覧処理

通常閲覧処理については、同時アクセス数を 60 まで増加させたが、いずれも 15.6 msec 程度で応答していることから、他の動的処理と比較してもサーバへの付加は少ないことがわかる（図 5.2）。

5.3.2　ログイン処理

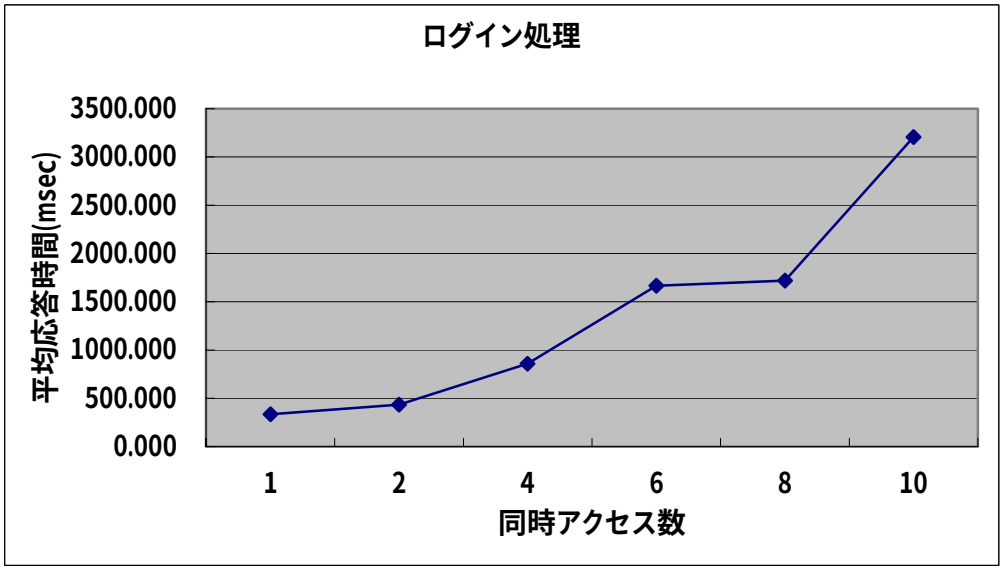


図 5.3　ログイン処理

ログイン処理では、1 処理あたり 300 msec 程度で応答している。通常閲覧処理と比較すると時間がかかっているが、これは動的コンテンツであるためと考えられる。また、同時接続数が増加するにつれ応答時間も増加している。今回は、表 3.2 に記載する各種サーバおよび、DBMS の設定は初期設定のままとなっている。従って、最大同時アクセス数等の変更によるサーバチューニングで改善されると考えられる（図 5.3)。

5.3.3　新規アカウント作成処理

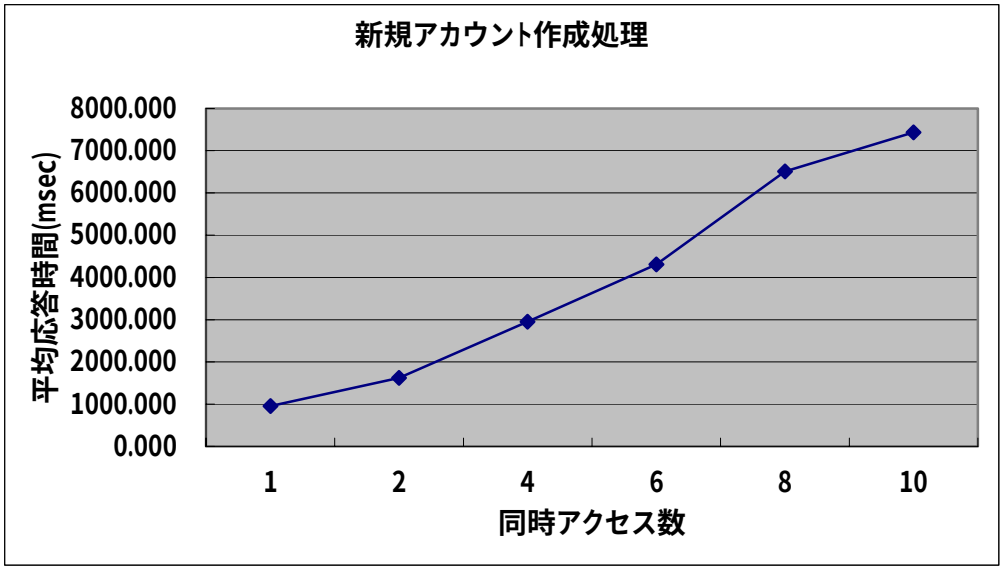


図 5.4　新規アカウント作成処理

新規アカウント作成処理では、１処理あたり 1000 msec 程度で応答している。同じ動的処理である、ログイン処理と比較しても時間がかかっている。これは、ログイン情報の発行やログイン画像のダウンロード、DBMS 接続数などがログイン処理に比べ多いためである（図 5.4)。

また、同時アクセス数の増加に伴い、平均応答時間は比例するように増加している。これは､トランザクションの衝突をさけているためである(3.3.3 節)｡これを裏付けるために、新規アカウント作成において同時アクセス数を 10 としたときの応答時間を処理順にグラフにした（図 5.5)。同時アクセスにもかかわらず、リクエストは１つずつ処理されていることがわかる。また、処理時間と待ち時間の差分をとれば、１処理あたり約 1000 msec で応答していることが見てとれる。

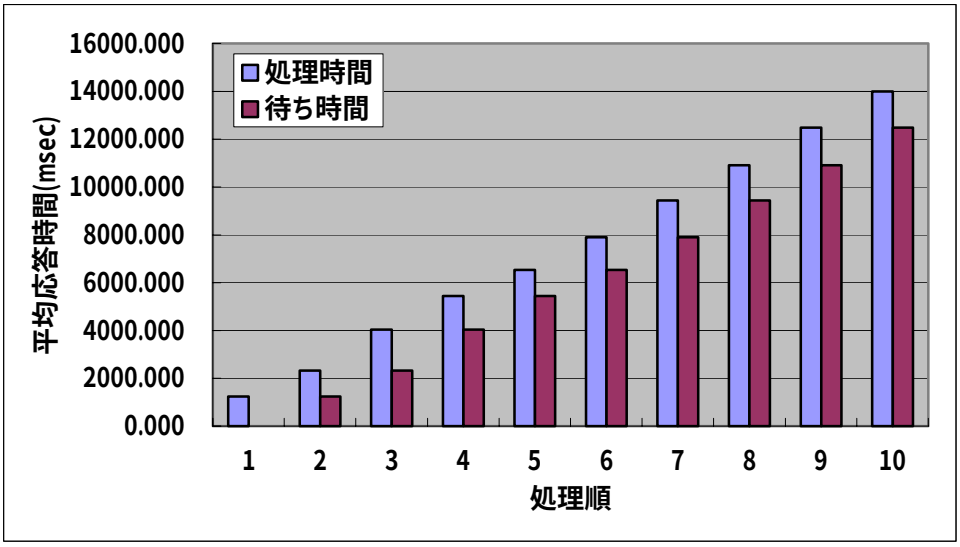
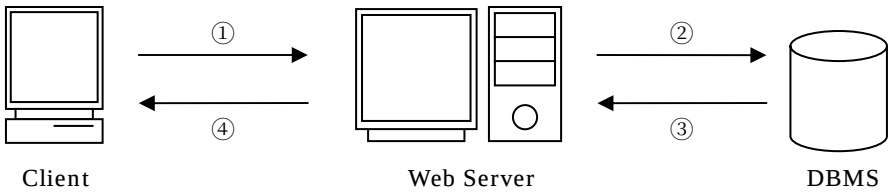


図 5.5　トランザクションの待ち時間

5.3.4　詳細測定

詳細測定は、同時アクセス実験とは異なり、Servlet を用いてサーバ内での処理時間を測定した。詳細測定での計測ポイントを以下に示す（図 5.6）。また、同時アクセスではなく、連続アクセスを行い、１処理時間の変動を見る。



計測時間　＝　②＋③

図 5.6　擬似ブラウザによる計測ポイント

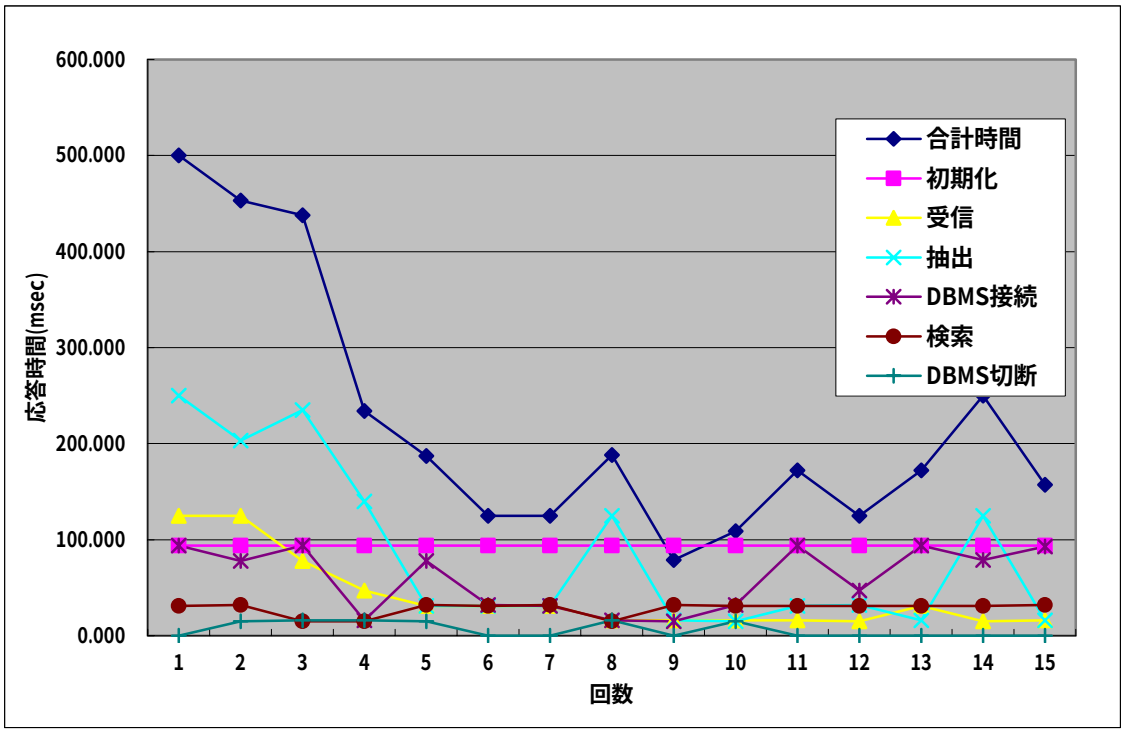


図 5.7　詳細測定（ログイン処理）

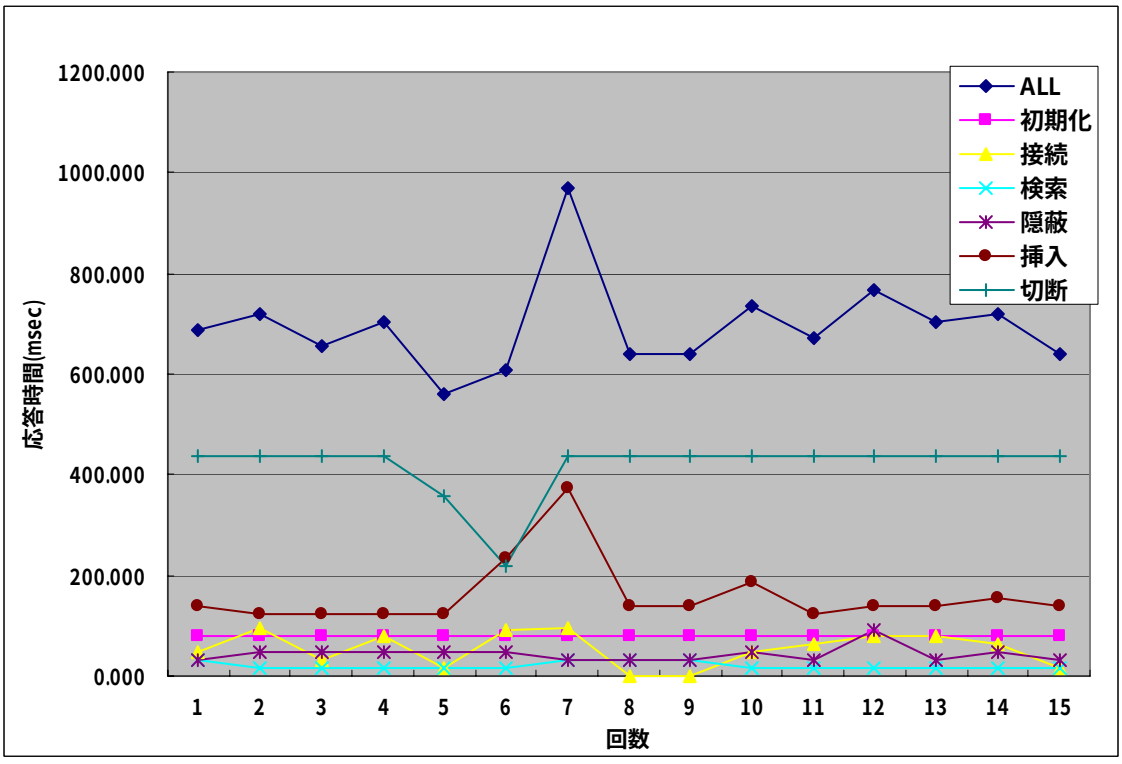


図 5.8　詳細測定（新規アカウント作成処理）

ログイン処理については、合計時間がログイン情報抽出時間と同様に変化していることが見て取れる。また、全体的にもログイン情報抽出時間が大きいことから、ログイン処理のボトルネックであると言える。また、処理回数を重ねる毎に処理時間が早くなる傾向にあり、この要因もログイン情報抽出時間となっている。この現象については後述する。

新規アカウント作成処理については、若干のばらつきが見られるものの、どの処理も一定の処理時間で応答している。また、ボトルネックは DBMS 切断処理であると言える。DBMS については同じ環境にも関わらす、ログイン処理と比較しても DBMS 切断処理時間が大きい。これは、JDBC によるトランザクション処理を行い、最後に手動コミットを行っているためであると考えられる。

ログイン処理について、処理回数を重ねる毎に処理時間が早くなる傾向の原因は Garbage Collection(GC)であると考えられる。GC は不要なメモリを見つけて自動的に開放する機能で、Java プログラマのメモリ管理の負担を軽減するために Java Virtual Machine(JVM)に標準装備されている。

そこで実験として、JVM の自動 GC に任せるのではなく、明示的にプログラム内で GC を行い、抽出処理の前に不要メモリを開放してログイン処理の詳細測定をおこなった（図 5.9）。

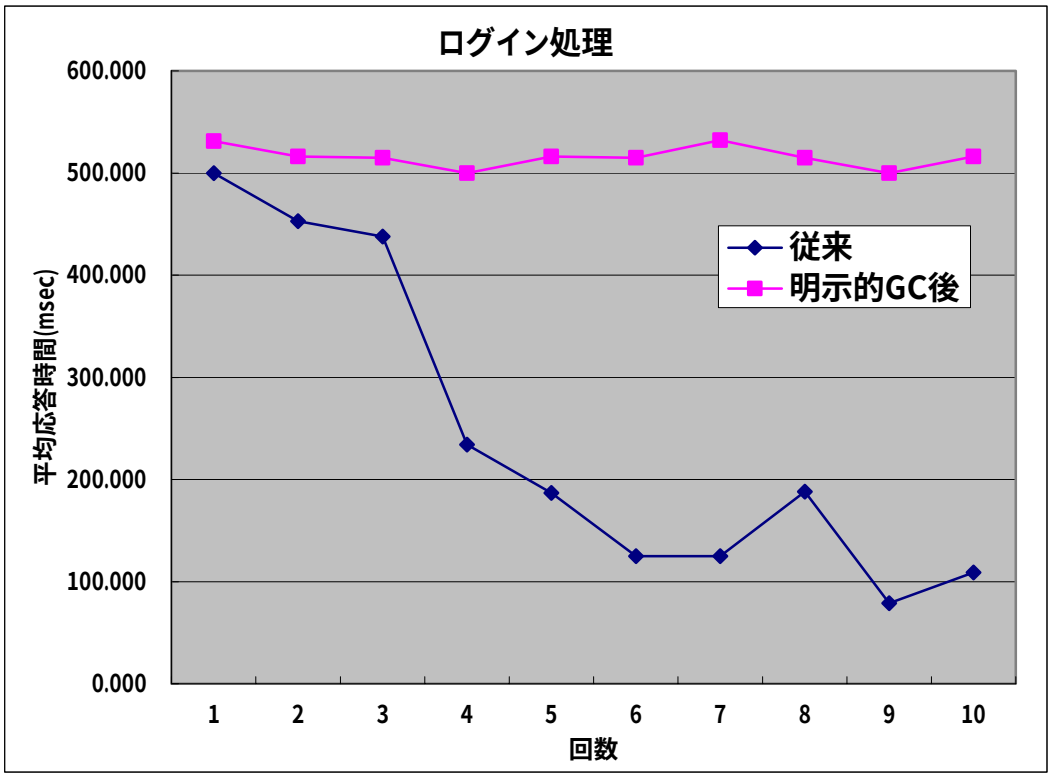


図 5.9　Garbage Collection に関する実験 1

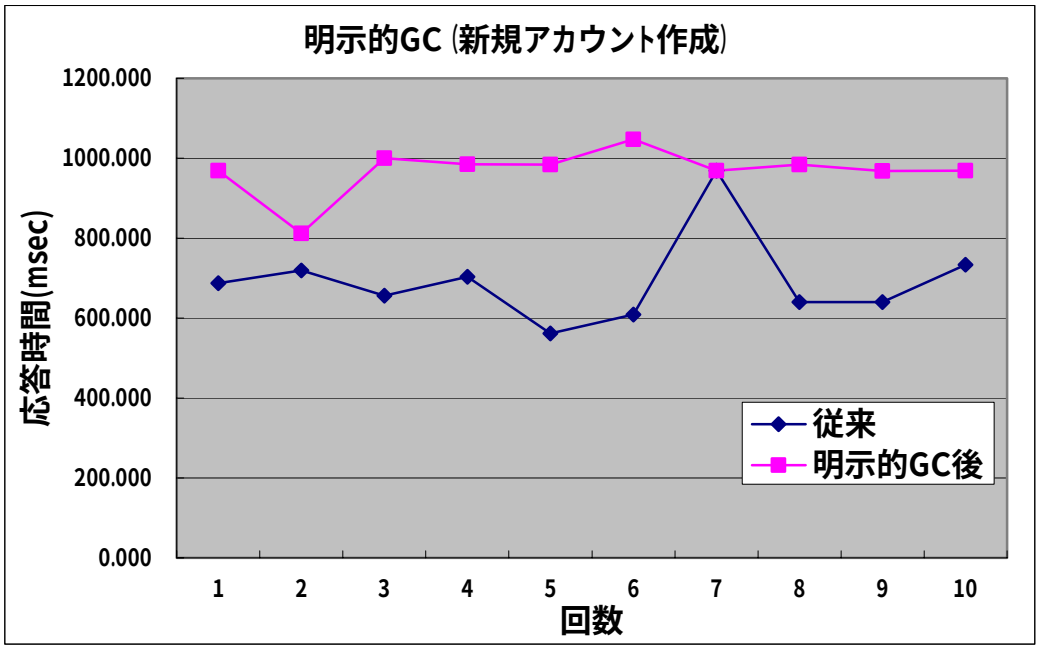


図 5.10　Garbage Collection に関する実験 2

図 5.9 から、処理回数を重ねる毎に処理時間が早くなる傾向は見られなくなった。また、データのバラつきも抑えられた。しかし若干データのバラつきは抑えられたものの、新規アカウント作成処理にみられなかった（図 5.10）。これは、JVM に装備されている GC 機能がオブジェクトの寿命を基準にする世代別 GC を採用しているためであると考えられる。そのため、処理によって早くなったり、データにバラつきがでたりと、異なった現象が現れると考えられるが、更なる究明が必要である。JVM に実装される GC 機能の詳細については http://java.sun.com/docs/hotspot/gc1.4.2/を参照されたい。また、今回の GC 実験は処理回数を重ねる毎に処理時間が早くなる傾向、およびデータのバラつき等の原因究明のために行ったもので、アプリケーションの最適化に勤めたものではない。

5.3.5　リクエスト成功率

これまでの実験において、応答時間とは別にリクエスト成功率を測定した。システムの構成上、リクエストが失敗時はエラーとしてクライアントに結果を返す。従って、ここでのリクエスト成功率とは、「正常にログインできる」「正常に新規アカウント作成ができる」ことを表す。以下にリクエスト成功率を示す（表 5.5）。

表 5.5　リクエスト成功率

実験項目	成功率	アクセスの種類
ログイン処理	58 %	同時アクセス
新規アカウント作成処理	100 %	同時アクセス
ログイン処理	100 %	連続アクセス
新規アカウント作成処理	100 %	連続アクセス

表 5.4 からログイン処理における同時アクセスのみ、リクエスト成功率 58%となっていることがわかる。また、同じ同時アクセスにも関わらず、新規アカウント作成処理ではリクエスト成功率 100%となっている。そこでリクエスト失敗時のエラー箇所を調査した（表 5.6）。

表 5.6　リクエスト失敗時のエラー

```
java.sql.SQLException:
 [HITACHI][HiRDB][HiRDB]KFPA11932-E
 Number of connect users exceeded max users
```

このエラー内容は、DBMS である HiRDB において、最大接続人数を超えているというものである。今回の実験とは別のマシンでいっていたとき、これほどリクエスト失敗率は見られなかった。（※正式な実験ではなかったため、残念ながら実験データが存在しない）わかっていることは、マシンスペックが今回の実験マシンよりも高かったことである。このことから、このエラーの原因を以下に示す（図 5.11）。

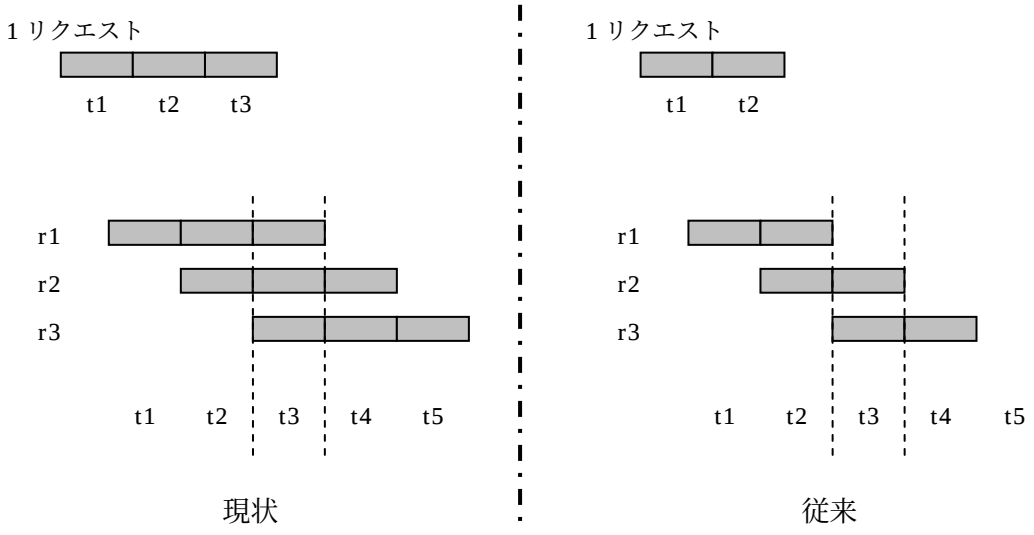


図 5.11 リクエスト失敗原因

いま、HiRDB の同時最大接続数を 2 とする。また、現在のサーバマシンはログイン処理１リクエストを 3t 時間、従来のサーバマシンは 2t 時間で処理できると考える。このとき、上図の動作を以下に示す（表 5.7）。

表 5.7　リクエスト失敗時の内部動作

Step	現状	従来
Step1	リクエスト r1 は正常に動作	リクエスト r1 は正常に動作
Step2	同時最大接続数：2 より リクエスト r2 も正常に動作	同時最大接続数：2 より リクエスト r2 も正常に動作
Step3	同時最大接続数：2 より リクエスト r3 は r1･r2 が DBMS に接続しているため接続エラー	同時最大接続数：2 より リクエスト r3 は r１がすでに終了し ているため正常に動作

つまり、Step3 において、マシンスペックの差からリクエスト失敗に繋がったと考えられる。ここで図 5.11・表 5.7 は例であり、今回のログインサーバが３つ目のリクエストで必ず失敗するというわけではない。また新規アカウント作成では、トランザクションの衝突をさけるための制御（3.3.3 節）がなされているため、複数のリクエストが同時に DBMS に接続することはなく、同じ同時接続にも関わらずリクエスト成功率は 100%となっていた。

この対策としてはマシンに合わせたサーバチューニングをする必要がある。その設定項目を以下に示す（表 5.8）。

表 5.8 サーバチューニング

設定箇所	内容
Tomcat	Data Source を用いて Tomcat でコネクションプーリング を使用しているため、タイムアウト時間や最大接続数など を変更するチューニング
HiRDB	HiRDB における同時最大接続数や最大応答待ち時間など を変更するチューニング

上記の 2 箇所の設定変更が考えられ、最適な設定を組み合わせる必要があるが、現状では実装されてなく、今後の課題となる。

第６章　結論

6.1 まとめ

本研究において、ステガノグラフィを用いてログイン情報を画像に埋め込み、ログイン画像をサーバに送信するだけでログインできるシステムを実装できた。結果として、ログイン情報暗記・入力の手間を省き、パスワード認証をベースとしているためユーザの心理的圧迫感も少ない。さらに、画像にログイン情報が隠蔽されていることを知る本人のみが利用できる。また、従来のパスワードに比べ、画像はログイン情報の管理を助けるシステムとなった。

RSA 暗号と比較してもログイン画像の頑健性は高く、RSA 暗号と併用することで、一層高い頑健性を得ることも可能であることが判った。また隠蔽方法の工夫により、ユーザの嗜好性を考慮し、ユーザが独自で有する画像を元にログイン画像を作成することも可能であることが判った。

実験では、クライアントからログインサーバシステムに擬似的に様々なアクセスを行うことで、サーバの動作を確認できた。さらに同時アクセスや連続アクセスでのリクエスト成功率を測定し、実装したログインシステムの現状での耐久性を把握できた。

6.2　今後の課題

今後の課題としては、ログイン画像の頑健性向上のため、埋め込み位置拡散を実装する必要がある。また実験の結果から、個々の処理におけるパフォーマンスを向上する必要もあるが、ログイン処理におけるリクエスト成功率を向上させるサーバチューニングが必要であることが判った。さらに、実際に想定される通常閲覧処理、ログイン処理、新規アカウント作成の３処理混合実験も行う必要がある。最終的な課題として、本システムは BMP 画像のみ対応しているが、より多くのユーザの嗜好性に合わせるために、一般的な JPEG 等その他のフォーマットへの対応も必要である。

謝辞

本研究にあたり、最後まで熱心なご指導をいただきました田中章司郎教授には、心より御礼申し上げます。

また、田中研究室の高木明先輩、長田昌訓君、学部生のみなさんには、本研究に関して数々の御協力と御助言をいただきました。厚く御礼申し上げます。なお、本論文、本研究で作成したプログラム及びデータ、並びに発表資料等の全ての知的財産権を、本研究の指導教官である田中章司郎教授に譲渡致します。

引用文献

[1] 辻井重男, 笠原正雄: 暗号と情報セキュリティ, p.249, 昭晃堂(1992).

[2] 小野束, 電子透かしとコンテンツ保護, p.260, オーム社(2001).

[3] Neil F. Johnson, Sushil Jajodia: Exploring Steganography : Seeing the Unseen, IEEE Computer pp.26-34 (February 1998).

[4] Neil F. Johnson, Sushil jajodia: Steganalysis of Images Created Using Current Steganography Software, LNCS, vol.1525, pp.273-289 (1998).

[5] Rajarathnam Chandramouli, Mehdi Kharrazi, and Nasir Memon: Image Steganography and Steganalysis : Consept and Practice, LNCS, vol.2939, pp.35-49 (2004).

[6] Jessica Fridrich, Miroslav Goljan, and Rui Du: Detecting LSB Steganography in Color and Gray-Scale Images, IEEE MultiMedia, pp.22-28 (October 2001).

[7] R. L. Rivest, A. Shamir, and L. Adleman: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM, vol.21, pp.120-126 (February 1978).