A. A short Java RMI tutorial

A.1 Java Remote Method Invocation

This is a technical literature study which purpose is to describe the basic parts of Java Remote Method Invocation.

Remote Method Invocation, abbreviated as RMI [SUN02, p2], provides support for distributed objects in Java, i.e. it allows objects to invoke methods on remote objects. The calling objects can use the exact same syntax as for local invocations [CDK01, p194].

The Java RMI model has two general requirements. The first requirement is that the RMI model shall be simple and easy to use and the second requirement it that the model shall fit into the Java language in a natural way [SUN02 ,p2].

A.2 Distributed Object Application

An RMI application is often composed of two separate programs, a server and a client [SUN02 ,p3]. The server creates remotes objects and makes references to those objects accessible. Then it waits for clients to invoke methods on the objects. The client gets remote references to remote objects in the server and invokes methods on those remote objects.

The RMI model provides an distributed object application to the programmer [SUN02 ,p3]. It is a mechanism that the server and the client use to communicate and pass information between each other. A distributed object application has to handle the following properties:

- **Locate remote objects:** The system has to obtain references to remote objects. This can be done i two ways. Either by using RMI's naming facility, the rmiregistry, or by passing and returning remote objects.
- **Communicate with remote objects:** The programmer doesn't have to handle communication between the remote objects since this is handled by the RMI system. The remote communication looks like an ordinary method invocation for the programmer.
- **Load class bytecodes for objects that are passed as parameters or return values:** All mechanisms for loading an object's code and transmitting data is provided by the RMI system.

Figure A-1, below, illustrates an RMI distributed application. In this example the RMI registry is used to obtain references to a remote object. First the server associates a name with a remote object in the RMI registry (see note 1 in figure A-1). When a client wants access to a remote object it looks up the object, by its name, in the registry (see note 2 in figure A-1). Then the client can invoke methods on the remote object (see note 3 in figure A-1) at the server.

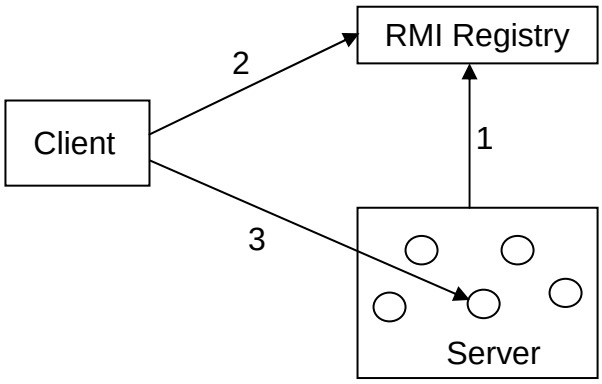

Figure A-1. An Illustration of a distributed object application.

A.3 Interfaces and Classes

Since Java RMI is a single-language system, the programming of distributed application in RMI is rather simple [CDK01, p194]. All interfaces and classes for the RMI system are defined in the *java.rmi* package [SUN02 ,p6]. Figure A-2, below, illustrates the relationship between some of the classes and interfaces. The *RemoteObject* class implements the *Remote* interface while the other classes extend *RemoteObject*.

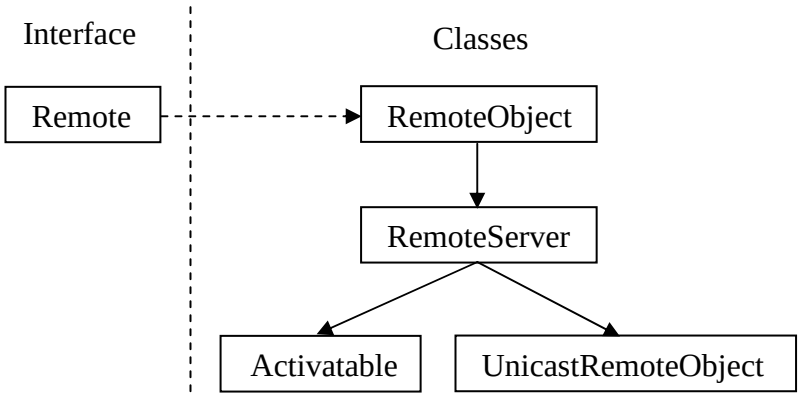

Figure A-2. Interfaces and Classes in the *java.rmi* package.

A.3.1 The Remote Interface

A remote interface is defined by extending the *Remote* interface that is provided in the java.rmi package. The remote interface is the interface that declares methods that clients can invoke from a remote virtual machine [SUN02 ,p6]. The remote interface must satisfy the following conditions:

- It must extend the interface *Remote*.
- Each remote method declaration in the remote interface must include the exception *RemoteException* (or one of it's superclasses) in it's thrown clause.

A.3.2 The RemoteObject Class

RMI server functions are provided by the class *RemoteObject* and its subclasses *RemoteServer*, *UnicastRemoteObject* and *Activatable*. Here is a short description of what the diffrent classes handle:

- RemoteObject provides implementations of the methods *hashCode, equals* and *toString* in the class *java.lang.Object*.
- The classes UnicastRemoteObject and Activatable create remote objects and export them, i.e. the classes make the remote objects available to remote clients.

A.3.3 The RemoteException Class

The class *RemoteException* is a superclass of the exceptions that the RMI system throws during a remote method invocation [SUN02 ,p6]. Each remote method that is declared in a remote interface must specify *RemoteException* (or one of it's superclasses) in it's throws clause to ensure the robustness of applications in the RMI system.

When a remote method invocation fails, the exception RemoteException is thrown. Communication failure, protocol errors and failure during marshalling or unmarshalling of parameters or return values are some reasons for RMI failure.

RemoteException is an exception that must be handled by the caller of the remote method, i.e. it is a checked exception. The compiler ensures that the programmer have handled these exceptions.

A.4 Implementation of a simple RMI system

This is a simple RMI system with a client and a server. The server contains one method (*helloWorld*) that returns a string to the client.

To build the RMI system all files has to be compiled. Then the stub and the skeleton, which are standard mechanisms communication with remote objects, are created with the *rmic* compiler [SUN02 ,p16].

This RMI system contains the following files (the files are shown below):

- **HelloWorld.java:** The remote interface.
- **HelloWorldClient.java:** The client application in the RMI system.
- **HelloWorldServer.java:** The server application in the RMI system.

When all files are compiled, performing the following command will create the stud and the skeleton:

rmic HelloWorldServer

Then the two classes will be created, *HelloWorldServer_Stub.class* and *HelloWorldServer_Skel.class*, where the first class represents the client side of the RMI system and the second file represents the server side of the RMI system.

HelloWorld.java

```
/*
  Filename: HelloWorld.java
*/

import java.rmi.Remote;
import java.rmi.RemoteException;

/*
  Classname: HelloWorld
  Comment: The remote interface.
*/
public interface HelloWorld extends Remote {
    String helloWorld() throws RemoteException;
}
```

HelloWorldClient.java

```
/*
  Filename: HelloWorldClient.java
*/

import java.rmi.Naming;
import java.rmi.RemoteException;

/*
  Classname: HelloWorldClient
  Comment: The RMI client.
*/
public class HelloWorldClient {

    static String message = "blank";

    // The HelloWorld object "obj" is the identifier that is
    // used to refer to the remote object that implements
    // the HelloWorld interface.

    static HelloWorld obj = null;

    public static void main(String args[])
    {
        try {
            obj = (HelloWorld)Naming.lookup("//"
                                            + "kvist.cs.umu.se"
                                            + "/HelloWorld");

           message = obj.helloWorld();

           System.out.println("Message from the RMI-server was: \""
                              + message + "\"");
        }

        catch (Exception e) {
            System.out.println("HelloWorldClient exception: "
                              + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

HelloWorldServer.java

```
/*
  Filename: HelloWorldServer.java
*/

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

/*
  Classname: HelloWorldServer
  Purpose: The RMI server.
*/
public class HelloWorldServer extends UnicastRemoteObject
    implements HelloWorld {

    public HelloWorldServer() throws RemoteException {
        super();
    }

    public String helloWorld() {
        System.out.println("Invocation to helloWorld was succesful!");
        return  "Hello World from RMI server!";
    }

    public static void main(String args[]) {

        try {
           // Create an object of the HelloWorldServer class.
           HelloWorldServer obj = new HelloWorldServer();

           // Bind this object instance to the name "HelloServer".
           Naming.rebind("HelloWorld", obj);

           System.out.println("HelloWorld bound in registry");
        }

        catch (Exception e) {
           System.out.println("HelloWorldServer error: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

B. A short PostgreSQL tutorial

B.1 PostgreSQL

PostgreSQL is an open source project that has been developed in various forms since 1977. It is considered to be the most advanced open source database system in the world since it provides many features that are traditionally seen only in enterprise-caliber commercial products [WJD02, p3].

PostgreSQL is an Object-Relational Database Management System (ORDBMS), which is an extension to the Relational Database Management System (RDBMS). A user can store, related pieces of data, in a two-dimensional data structure called tables. The data in the table may consist of many different types of data, e.g. integers, floating point numbers and strings [WJD02, p35].

A table is composed of *columns* and *rows*. The intersection between a column and a row is called a *field*. A column, within a table, describes the name and the type of the data that will be found in a row for that column's field. A row, within a table, represents a record that is composed of fields that are described by their corresponding column's name and type [WJD02, p35]. Table B-1, below, gives an example of a SQL table for a telephone directory.

id	first_name	last_name	telephone_number
254	John	Worsley	090-123456
34	Joshua	Drake	0920-12345
1354	Patrik	Andersson	0978-56789
3006	George	Baker	031-112233
891	Tony	Adams	08-12345678
99	William	Ross	018-1234567

Table B-1. An example of an SQL table.

B.2 SQL Statements

An SQL statement begins with a *command*. The command is a word or a group of words that describes what action the statement will initiate. Table B-2, below, describes some basic SQL commands [WJD02, p39].

Command	Description
CREATE DATABASE	Creates a new database (this command is used by the administrator of the system)
DROP DATABASE	Destroys an existing database (this command is used by the administrator of the system).
CREATE TABLE	Create a table in an existing database.
DROP TABLE	Destroys an existing table.
SELECT	Retrieves records from a table.
INSERT	Adds records into a table.
UPDATE	Modifies the data in existing table records.
DELETE	Removes existing records from a table

Table B-2. Some fundamental PostgreSQL commands.

The *command*, in an SQL statement, can be called verb of the statement, since it always describes the action to be taken. A statement also consists of one or more *clauses*, which describes the function of the SQL statement.

When SQL was designed, the ease of use and readability was an important aspect of the design. Because of the readability aspect, SQL statements have a strong resemblance to simple English sentences. When you are reading a well-designed SQL query it should be almost as easy as reading an ordinary sentence.

B.3 Using JDBC to communicate with PostgreSQL

Java Database Connectivity, abbreviated as JDBC, is a set of classes and methods that covers all the interaction you can have with a standard SQL database. Using JDBC with Java is a simple and portable way of interacting with different types of databases [WJD02, p433].

B.3.1 JDBC Driver

The driver that PostgreSQL provides is a so-called Type 4 driver. This means that the driver is written in pure Java. Since the driver is written in pure Java, it can be taken anywhere and be used anywhere as long as the platform has TCP/IP capabilities (because the driver only connects with TCP/IP). A precompiled version of the driver can be downloaded from PostgreSQL JDBC web site[1] [WJD02, p434].

B.3.2 Registration of the driver

To be able to use the driver, you have to ensure that the driver gets registered in your code. The method *Class.forName* finds a class by its name. If the class is found, the loader will read in the binary description of the class.

[1] http://jdbc.postgresql.org/

Then the *Driver* class registers itself, with the *DriverManager* class, when it passes through the Java class loader. When the register is done JDBC will know what driver to use when connecting to a specific database [WJD02, p435].

B.3.3 Connecting to a database

When the *Driver* class is registered, you can request a connection to a PostgreSQL database. The class *DriverManager* is used to set up a connection. This class is responsible for handling JDBC URL's, finding the appropriate driver, and using that driver to set up a connection to the database [WJD02, p435].

The JDBC URL, which shall be used, when connecting to a database has the following format:

jdbc:[drivertype]:[database]

The first part of the URL, *jdbc*, is a constant that indicates that this connection is a connection to a JDBC data source. The *drivertype* indicates which kind of database you want to connect to, in this case *postgresql*. The last part, of the URL, is the *database*. This part is passed off to the driver, which finds the database.

B.3.4 Basic JDBC usage

The *Connection*, *Statement* and *ResultSet* classes represent some of the basic interaction, which programmers ask for, with SQL. A *Connection* object represents a physical connection to a database. The *Connection* object can be used to create a *Statement* object, which is JDBC's way of getting SQL statements to the database [WJD02, p438].

The *ResultSet* class is the primary interface for fetching information from the database. These objects are returned from SQL statements the have been executed. A *ResultSet* object can step through the rows returned and it can return the value of a specific column in a row.

The *Statement* class has two important methods, *executeQuery* and *executeUpdate*. The *executeQuery* method takes the SQL statement, which shall be executed, as argument and returns a *ResultSet* object. It is used when executing queries that will return a set of data, e.g. a SELECT statement. The *executeUpdate* method also takes the SQL statement, which shall be executed, as argument. The different between these two methods is that *executeUpdate* is for executing statements that change the data in the database, e.g. CREATE, INSERT or UPDATE statements. It returns an int that corresponds to the number of records that were modified.

B.3.5 A simple JDBC application

This is a simple JDBC application that creates a telephone book and adds some element (i.e. some persons) to the table. Then it fetches the elements from the tables and prints them on the screen. At least the table is dropped from the database.

This application contains only one file, *DatabaseTest.java*, which is shown below. To be able to test this application, you must have a *postgresql* database installed on your system.

DatabaseTest.java

```
/*
  Filename: DatabaseTest.java
*/

import java.sql.*;

/*
  Classname: DatabaseTest
  Comment: A testapplication that uses JDBC to access the PostgreSQL
server.
*/
public class DatabaseTest {

    public static void main(String args[])
    {
        int element=1;

        try {

            // Loading the Driver
            try{
                Class.forName("org.postgresql.Driver");
            }catch(ClassNotFoundException cnfe){
                System.err.println("Couldn't find driver class:");
                cnfe.printStackTrace();
            }

            // Get a connection to the PostgreSQL database.
            Connection db =  DriverManager.getConnection
            ("jdbc:postgresql://postgres.cs.umu.se/ei97pva", "ei97pva",
            null);


            System.out.println("The connection to the database was
            succesfully opened.");

            // Get a Statement instance.
            Statement st = db.createStatement();

            // Create a table named telephonebook
            st.executeUpdate
            ("CREATE TABLE telephonebook ( first varchar(30), last
            varchar(30), number varchar(20) );");
```

```
            // Insert some elements in the database
            st.executeUpdate
            ("INSERT INTO telephonebook VALUES ('Johan', 'Johansson', '090-
            123456');");

            st.executeUpdate
            ("INSERT INTO telephonebook VALUES ('Sven', 'Svensson', '090-
            654321');");

            st.executeUpdate
            ("INSERT INTO telephonebook VALUES ('Anders', 'Andersson',
            '090-456789');");


            st.executeUpdate
            ("INSERT INTO telephonebook VALUES ('Per', 'Persson', '090-
            987654');");

            //Execute a query to get all elements in the table.
            ResultSet rs = st.executeQuery("SELECT * FROM telephonebook");

            // Print out all elements in the table named telephonebook.
            while( rs.next() ) {

                System.out.print("Element " + element + ": ");
                System.out.print(rs.getString(1));
                System.out.print(" ");
                System.out.print(rs.getString(2));
                System.out.print(" : ");
                System.out.println(rs.getString(3));

                element=element+1;
            }

            // Drop the table named telephonebook.
            st.execute("DROP TABLE telephonebook");

            // Close the ResultSet and the Statement variables and close
            // the connection to the database.
            rs.close();
            st.close();
            db.close();

            System.out.println("Closed connection to the database.");
        }

        catch (Exception e) {
            System.out.println("DatabaseTest exception: "
                               + e.getMessage());
            e.printStackTrace();
        }

    }
}
```

C. Sequence Diagrams

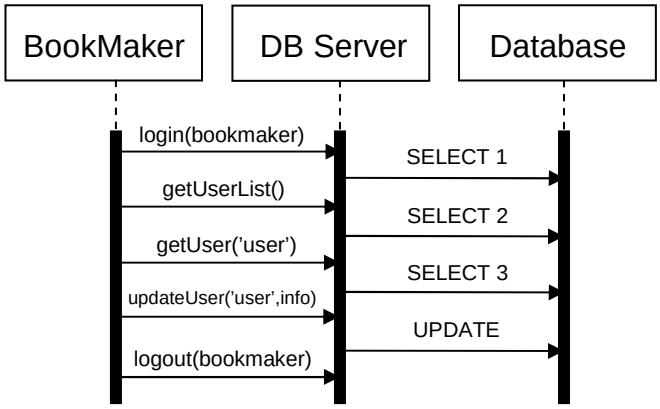

Figure C-1. The bookmaker updates user information.

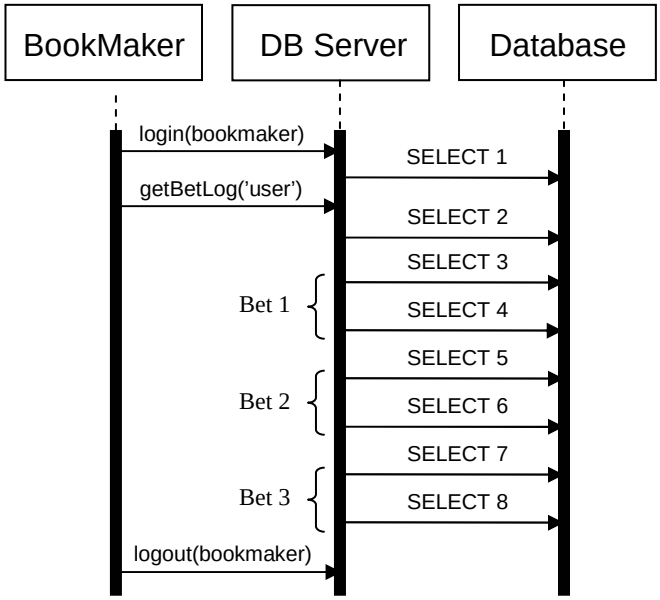

Figure C-2. The bookmaker view bet logs.

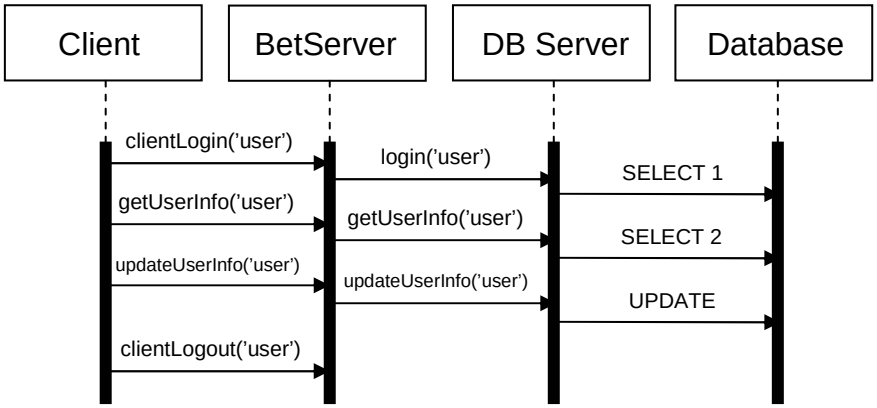

Figure C-3. A client logs on and updates its user information.

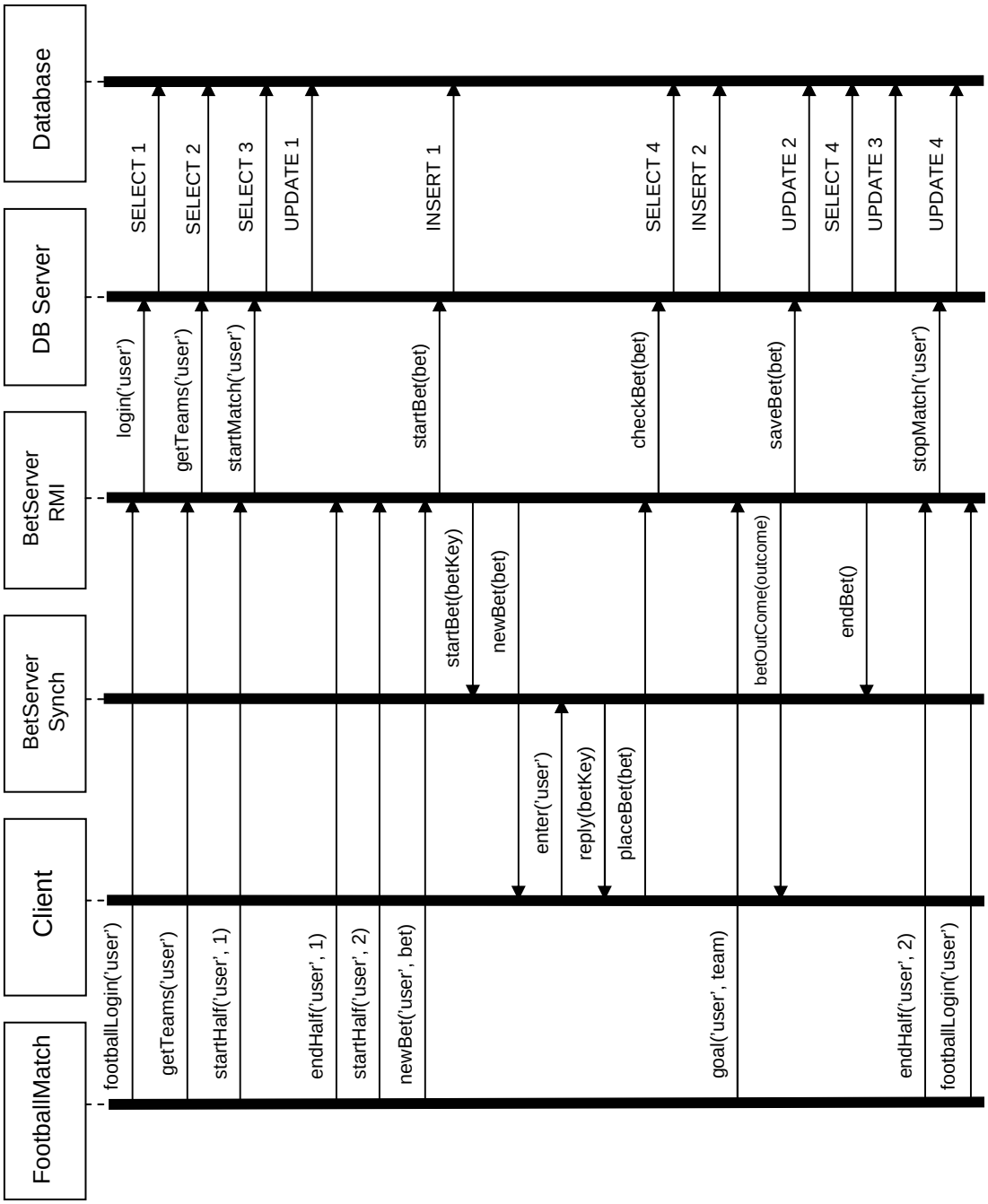

Figure C-4. A simulation of a football match.

D. Screenshots over the GUI's

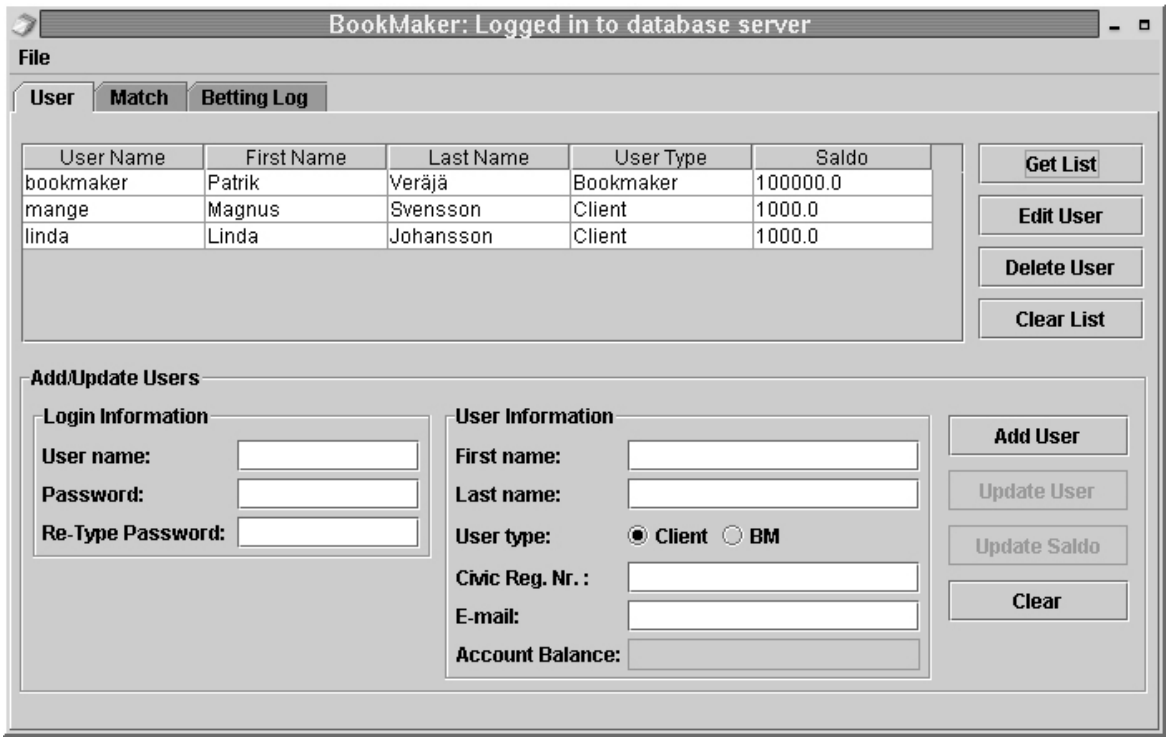

Figure D-1. The User pane in the Bookmaker GUI.

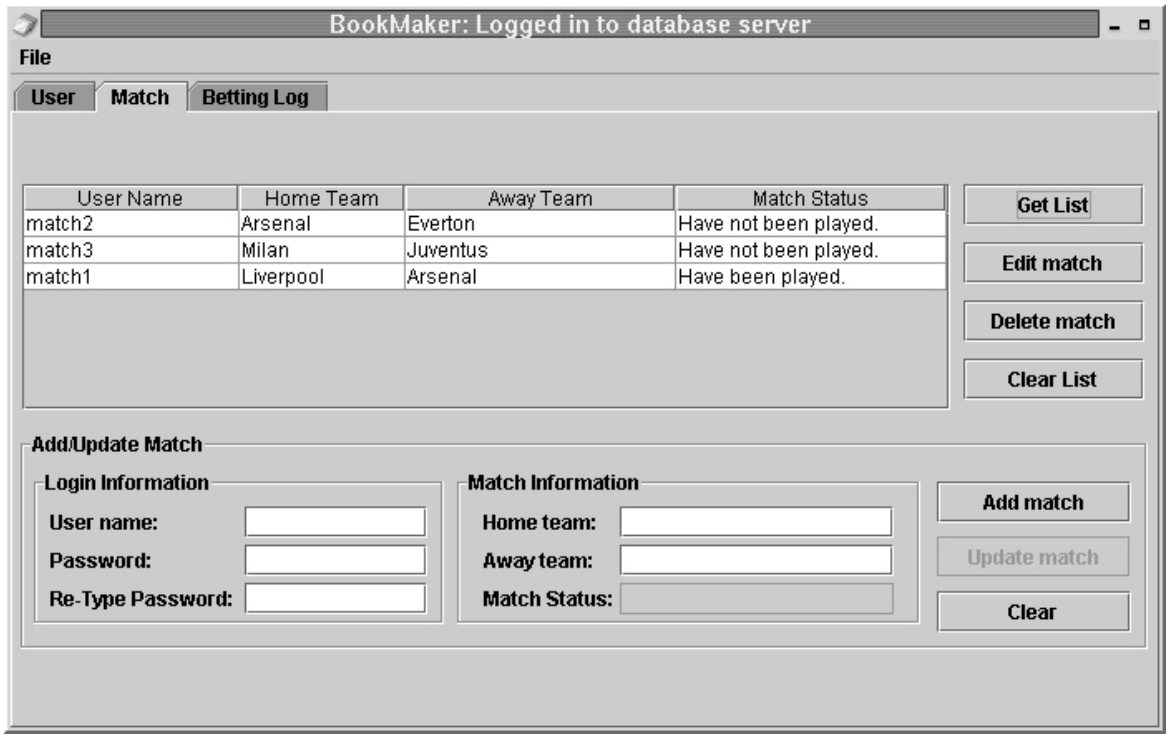

Figure D-2. The Match pane in the Bookmaker GUI.

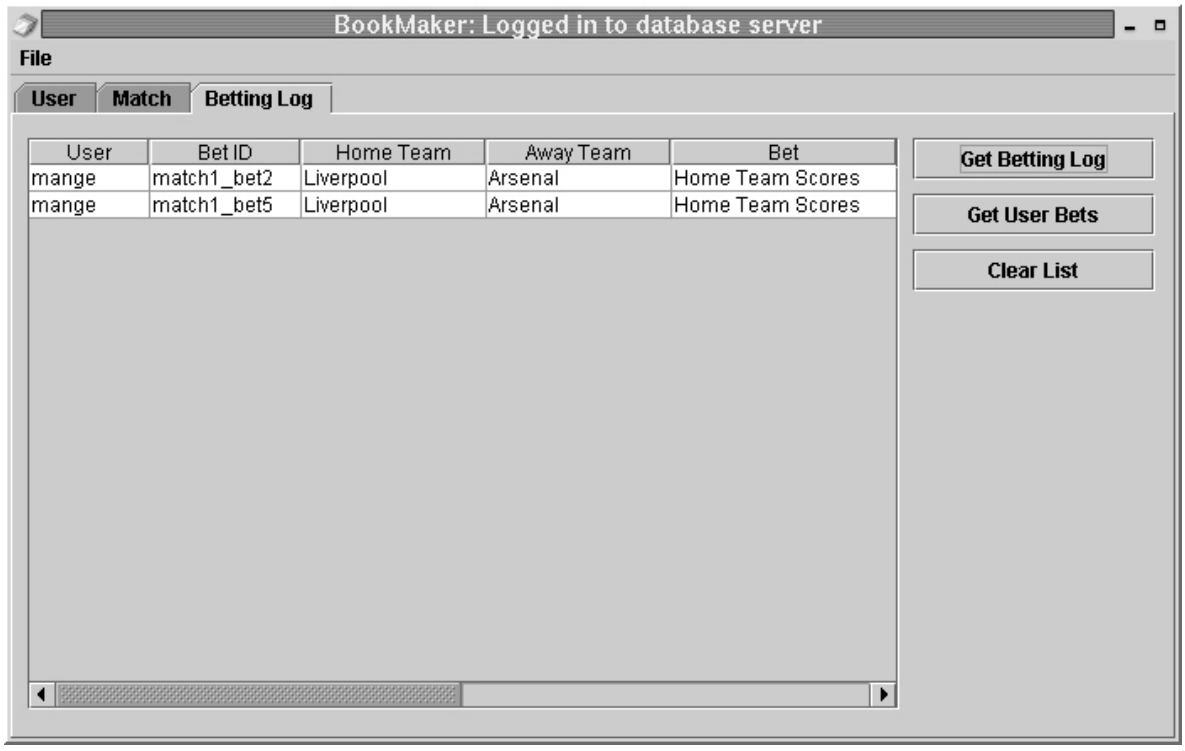

Figure D-3. First half of the bet log table in the Bookmaker GUI.

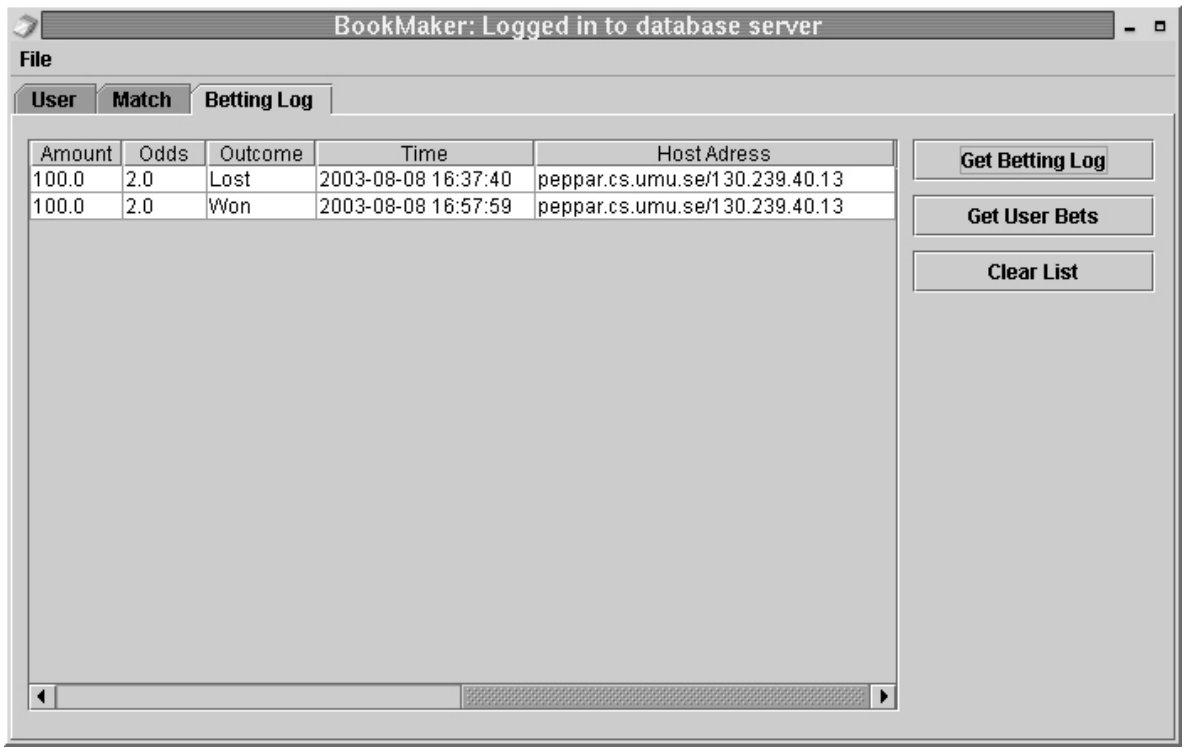

Figure D-4. Second half of the bet log table in the Bookmaker GUI.

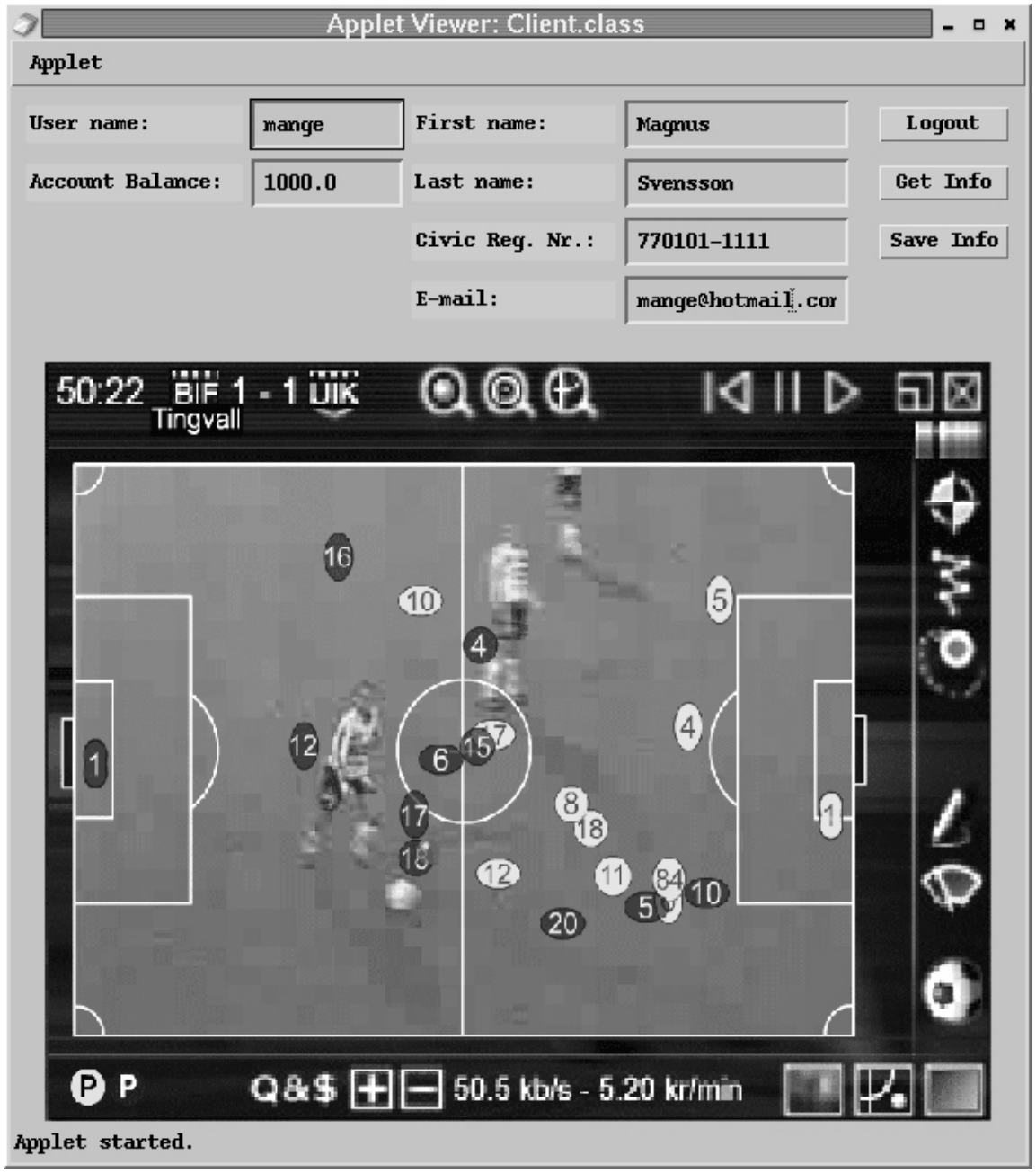

Figure D-5. Screenshot over the client applet GUI.

Figure D-6. Screenshot over the football match GUI.

E. The Evaluation Questionnaire

Evaluation Questionnaire

This is an evaluation questionnaire which primary purpose is to determine if this thesis has managed to present the concept behind rapid betting. A secondary purpose is to determine if rapid betting is an interesting form of betting.

1. Rate your sports interest (1-5): ……

2. Which sports are you most interested in? ……..
……..

3. How often do you use to bet? **Often** …. **Sometimes** …. **Never** ….

4. What do you use to bet on?

Dogs	……
Harness Racing	……
Ice hockey	……
Football	……
Floor ball	……..
Bandy	……
Handball	……
Volleyball	……
Other sports	……..

5. Do you use to bet in your local kiosk? **Yes** ….. **No** ……

6. Do you know any of these betting addresses on the Internet?

www.atg.se	…….
www.betandwin.com	…….
www.centrebet.com	…….
www.expekt.com	…….
www.ladbrokes.com	…….
www.svenskaspel.se	…….
www.unibet.com	…….
Other web sites	…….

7. Do you use to be on any of these sites?

……

8. Do you understand the concepts behind rapid betting?

……
……

9. Would you use a system for rapid betting (i.e. on your mobile device)?

……
……

10. Other comments

……
……