# A prototype of a distributed system for rapid betting

20p Master Thesis
Department of Computing Science
Umeå University

By
Patrik Veräjä
ei97pva@cs.umu.se

Umeå, 2003

Tutor: Greger Wikstrand (greger@cs.umu.se)
Examiner: Per Lindström (perl@cs.umu.se)

# Abstract

In February 2002, Ericsson and the Umeå Center for Interaction Technology (UCIT) started a project which purpose was to investigate the best way to visualize a match through video and animations on a mobile device. During the project a thought came up. In the future, if it will be possible to view a football match on mobile devices, some users may want to be able to place bets during the match. E.g. place bets on whether the teams scores within a certain time or not. In this thesis this concept is called *rapid betting*.

The purpose with this thesis was to implement a prototype of a distributed system for *rapid betting*. Synchronization was an important part of this thesis, since it is important that the user cannot be able to place bets after the bet has expired.

Therefore, a literature study was performed to find an appropriate synchronization algorithm to use in the implementation of the betting system. The algorithms that were investigated in this thesis are Cristian's method, a token passing approach and a centralized approach.

Java RMI and PostgreSQL are important tools in the implementation of the betting system. Therefore, tutorials are included in this thesis so readers, within no experience of these tools, can get a short introduction to these tools.

# Keywords

rapid betting, synchronization, physical clocks, mutual exclusion, Java RMI, PostgreSQL, JDBC

# Sammanfattning på svenska

Detta är ett examensarbete som utfördes på institutionen för Datavetenskap vid Umeå Universitet under våren och sommaren 2003.

I februari 2002 startade Ericsson och Umeå Center for Interaction Technology (UCIT) ett projekt vars syfte var att undersöka vilket som är det bästa sättet att, med video och animationer, visualisera en fotbollsmatch på en mobil enhet. Under arbetets gång föddes tanken att det kanske skulle kunna vara intressant för en del användare att, under en match, kunna lägga vad på om lagen gör mål inom en viss begränsad tid. Detta begrepp benämndes *rapid betting.*

Syftet med detta examensarbete var att implementera en prototyp till ett distribuerat system för *rapid betting* under en fotbollsmatch. En viktig aspekt vid designen av systemet var att en användare inte skall kunna lägga vad efter att dess giltighet har löpt ut. Därför gjordes ett fördjupningsarbete vars syfte var att finna en lämplig algoritm för att hantera synkroniseringen mellan klienterna och server.

I fördjupningsarbetet studerades tre stycken algoritmer. Dessa algoritmer var Cristian's metod (för synkronisering av fysiska klockor) samt en token-ring algoritm och en centraliserad algoritm för ömsesidig uteslutning till en krisisk sektion. Dessa algoritmer presenteras och jämförs med varandra för att komma fram till vilken algoritm som är mest lämplig att använda vid implementeringen av systemet.

Systemet består av tre stycken aktörer, bookmakern, fotbollsmatchen och klienterna. Bookmakern är administratören av systemet. Dessa användare lägger till och administrerar fotbollsmatcher och klienter. Fotbollsmatchen är en användare som simulerar en fotbollsmatch genom att generera händelser, med hjälp av ett GUI, som innehåller knappar för de olika händelserna. Klienterna är användare som lägger vad under en fotbollsmatch.

Rapporten innehållet även två tekniska introduktioner för de två verktyg, Java RMI och PostgreSQL, som är centrala vid implementeringen av systemet. Dessa två introduktioner bifogas som bilaga för att läsare, som inte har någon erfarenhet av dessa verktyg, snabbt skall kunna sätta sig in i de grundläggande delarna av dessa verktyg och därmed lättare kunna förstå systemdesignen och systemöversikten.

Som avslutning på examensarbetet utfördes en kort utvärdering, vars syfte var att ta reda på om examensarbetet har lyckats presentera begreppet *rapid betting* och konceptet bakom begreppet. Utvärderingen utfördes genom att några personer fick testa systemet. Sedan fick de svara på några frågor i en kort enkät. Resultatet av den korta utvärderingen visade att examensarbetet har lyckats presentera konceptet bakom rapid betting. Dock måste en större undersökning göras för att ta reda på om ett sådan här system skulle kunna vara kommersiellt intressant.

*Intressanta nyckelbegrepp i denna rapport är*: rapid betting, synkronisering, ömsesidig uteslutning, PostgreSQL, JDBC, Java RMI.

# Contents

# Figures



# Tables

# 1. Introduction

## 1.1. Background

In the future, when hardware will be improved, it might be interesting to view sport events live at a cellular phone. One project that researches this area is the Arena research project[1]. The Arena project is a project within the Mäkitalo Research Centre[2] that is an international research centre for wireless technology and applications. More about this project can be read at the websites in the footnotes.

The BASTARD[3] project, at the University of Umeå, is another project that researches the possibility to view sport events live through a cellular phone. The project and its progress are described below.

During the progress of the Bastard project an idea came up. Maybe some users want to be able to bet if the teams, in a sports event, scores within a certain time. This concept was called *rapid betting*.

## 1.2. The BASTARD project

In February 2002, Ericsson and the Umeå Center for Interaction Technology (UCIT), formulated a project which purpose was to investigate the best ways to visualize a match through video and animations. The project would also investigate how to enhance the user experience through interactivity. The project also had two main design goals. The first goal was to minimize the cost of the service and the second goal was to maximize the user experience [WEÖ03].

The project was divided into two phases. In the first phase the purpose was to find out whether animations or video was the best way to visualize the football match. The animations and the video were compared in an experiment where the participants were presented short clips from a football match. These clips were encoded in three different ways, either as animations, low bandwidth video or high bandwidth video. The participants watched these clips and then they answered questions. The result of this experiment was that neither animation nor video was better. Both methods had their advantages and disadvantages.

Before the second phase a workshop was held to find ways to proceed with the project. Out of this workshop came several ideas that could be investigated in the second phase. After this workshop a prototype was created. This prototype was tested on two groups. Based on the feedback that the study received the prototype were adjusted and tested again at the groups.

The study showed that the users wanted more interaction and that they wanted more data about the players. In the most recent prototype, several of the results and ideas from the two iterations have been incorporated. I.e. the users can select between video and animations and annotations are available to make the football game more understandable.

[1] www.cdt.luth.se/projects/arena/
[2] www.makitaloresearch.com
[3] Bandwidth Adapted STreaming Application Research and Demo

The project has not produced a product that is ready to be a commercial product. More ideas have to be gathered and these ideas should, together with the result from the project, be used to build a commercial product. The product shall be able to display data in the way that the user prefers and be able to receive streaming video and position data.

## 1.3. Project description

When more is known about a sports event, e.g. exactly where actors are, it will be possible to improve and enhance betting in several ways. For instance it might be possible to bet if the current attack, in a football match, will lead to a goal within a time limit.

As described above, a project group at the Umeå University is developing a prototype to a system for viewing football matches at a mobile phone. They also want a system that supports *rapid betting* on the football matches shown at the mobile phone.

The purpose with this master thesis is to develop a prototype of a distributed system for rapid betting on a football match. The betting system shall be implemented so it will be possible to integrate the betting system with the system for viewing football matches at a mobile phone, which is described above.

An important keyword for this thesis is synchronization, since it is important that the components in the system are synchronized. Therefore, a literature study is performed, which purpose is to find an appropriate synchronization algorithm for the implementation of this betting system.

## 1.4. The structure of this thesis

In chapter 2, the requirements on the betting system are presented. First some scenarios are presented and then the requirements, and the priority level for each requirement, are presented.

In chapter 3, the literature study is presented. The purpose with the literature study is to establish which, of the chosen algorithms, that is the most appropriate to use in this implementation.

In chapter 4 and 5, the system is presented. Chapter 4 is the system design that describes how the system is built. Design choices and restrictions are presented. Then all components are described. Finally it is described how the components communicate with each other. Chapter 5 is the system overview and it describes the classes in each component and how they are related to each other. In this chapter some sequence diagrams, for some typical actions, are presented. These diagrams show how the communications between the components are performed.

In chapter 6 and 7, the thesis is summarized. Chapter 6 presents a short evaluation which purpose is to determine how well the system was built and if it presents the rapid betting concept in a natural way. Chapter 7 closes the thesis by presenting the conclusions. It also presents some objects that must be improved if the system shall be a commercial product.

# 2. Requirements

This chapter described the requirements that were stated in the beginning of the thesis. It also describes some hypothetical scenarios and the actors in the betting system.

## 2.1. Hypothetical Scenarios

Liverpool plays a football match against Arsenal.

1. Arsenal is attacking Liverpool and a defender in Liverpool breaks the attack.
2. Liverpool dominates the match, i.e. Liverpool has the ball most of the time.

In these situations it shall be possible to place a bet that Liverpool will make a goal within a certain time limit.

In this thesis the focus will be on implementing the first scenario. The second scenario will be implemented only if there is adequate time in the end of the project.

## 2.2. The betting system

This section describes the actors in the betting system (see Figure 2-1), which has three groups of actors. The administrators, or bookmakers, administrates the system and create new bets. The football matches are users that simulate a football match. These users can start new bets and generate goals. The clients are user that places bets during a football match.

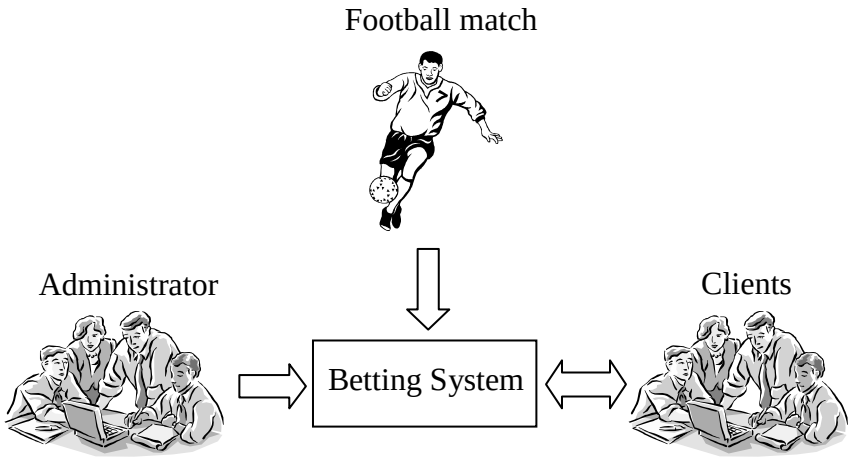


**Figure 2-1. Actors in the Betting System.**

## 2.3. Priority levels

The rest of this chapter list the requirements that were stated. There are three priority levels (1-3), where 1 has the lowest priority and 3 has the highest priority. The priority level for a requirement is presented within parenthesis after the requirement.

## 2.4. General requirements

a) The components in the system shall be synchronized, i.e. it shall not be possible to place a bet if a team already has scored (3).
b) All communication shall be done in a secure way (1).

## 2.5. Functional requirements

### 2.5.1. Administrator requirements

*The administrator shall be able to:*

a) log on to the system (3).
b) add new users and administrate existing users (3).
c) add new matches and administrate existing matches (3).
d) deposite and withdraw credits from the clients account (3).
e) start new bets (3).

### 2.5.2. Client requirements

*The clients shall be able to:*

a) log on to the system (3).
b) be notified of new bets, place a bet and get feedback about the outcome of the bet (3).
c) get the balance for his account (2).

## 2.6. Non-Functional Requirements

### 2.6.1. Administrator requirements

a) The time from when a bet is generated to when it is closed shall be 20 seconds (3).
b) The betting system shall handle only one active match (2).

### 2.6.2. Client requirements

a) The client application shall have a background picture that is a screenshot from the prototype of the football system (1).
**b)** The system has to be able to handle more than one client (3).

# 3. Synchronization in Distributed Systems

This chapter is a literature study which primary purpose is to investigate different algorithms for synchronization in distributed systems. These algorithms will be presented and compared. Then it will be determined which algorithm that is the most appropriate to use in the implementation of the betting system.

Accountability is an important security aspect in real-time systems. Unfortunately, accountability is a neglected aspect in security matters, since security typically focuses overly on e.g. confidentiality, integrity, strong authentication and authorization. It is also important to consider policies and mechanisms for accountability [NPG03].

It is especially important to consider accountability in a betting system, where it is important to know whether a placed bet is valid or not. Therefore, synchronization is an important part of this thesis, because the system must be able to determine if a bet was placed before the bet has expired.

## 3.1. Time

There are many reasons to why time is an important and interesting aspect in distributed systems. Here are two of the most important reasons [CDK01, p386].

One reason is that time is a very important quantity that we always want to measure accurately. Many distributed applications require that events can be timestamped so it will be possible to know what time of the day an event occurred at a particular computer. Therefore it must be possible to synchronize clocks with an external source. One example is a bank transaction. In bank transaction it is important, for auditing purpose, that event can be time stamped correctly.

Another reason is that many algorithms have been developed to solve distributed problems. These algorithms depend upon clock synchronization. Examples on such algorithms are these who include maintaining the consistency of distributed data (e.g. bank transactions), algorithms that check the authenticity of a request that is sent to a server (e.g. Kerberos) and algorithms that are eliminating the processing of duplicate updates.

### 3.1.1. Synchronization of physical clocks

There are two methods to synchronize physical clocks in a distributed system [CDK01, p389]. External synchronization is used when it is necessary to know what time of day events occurs at processes in a distributed system. In this method the processes clocks is synchronized with an external source of time.

In internal synchronization the processes clocks are synchronized to each other with a fixed degree of accuracy. This means that it is possible to measure intervals two events occurring at different computers

### 3.1.2. Cristian's method

This is a method that uses a time server to synchronize computers [CDK01, p391]. When a client wants to synchronize its clock, it sends a request $m_r$ to the time server (see figure 3-1). The time server supplies the time in a reply, $m_t$, to the client.

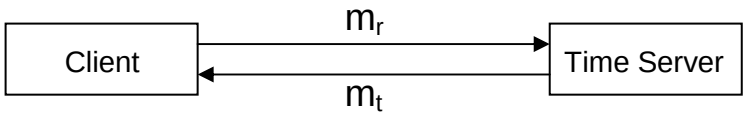


**Figure 3-1. An Illustration of Cristian's method for synchronizing clocks.**

The client record the round-trip time (RTT), which is the time, it takes from the client sends the request until it receives a reply from the time server. When the client has the RTT it can make a simple estimate of the time it should set its clock. A simple approach is to set the clock to t + RTT/2 (where t is the time that it receives in the reply from the time server). This approach assumes the time before the value of t is placed in $m_t$ is equally to the time before the time t is placed in the reply, which is normally a reasonable assumption.

The idea is that the clients and the betting server shall synchronize its clocks to the time server. When a client places a bet, it's will timestamp the bet and then it will send the bet to the betting server. On receiving a bet from a client, the betting server will check if the bet is valid by checking the timestamp and compare to its clock. Then it will decide whether the bet is valid or not.

There are some problems with Cristian's method. One problem is that a single server implements all service. If the server fails synchronization of time is impossible. To solve this problem Cristian suggested that the system could have a group of synchronized time servers. The clients could then multicast the request to the group and use the first reply that is obtained.

On other problem is that a faulty time server that replies with wrong time or an imposter time server (that replies with deliberately false time) could wreck the system.

## 3.2. Mutual exclusion algorithms

### 3.2.1. A centralized approach

In this approach, one process in the system is chosen to coordinate the entry to the critical section [SGG02, p598]. Three messages are required when a process wants to enter the critical section. These three messages are called *request*, *reply* and *release*.

When a process wants to enter the critical section it sends a *request* message to the coordinator. On receiving a *reply* from the coordinator, the process can enter the critical section. After the process has leaved the critical section it sends a *release* message to the coordinator.

When the coordinator receives a *request* message from a process it checks if any other process is in the critical section. The coordinator sends back a *reply* message immediately if no process is in the critical section. If another process is in the critical section the *request* is queued. On receiving a *release* message, the coordinator removes a *request* from the queue and sends a *reply* message to that process.

Since one coordinator handles the critical section, this approach ensures mutual exclusion to the critical section. If the system has a scheduling policy that is fair, then this approach also guarantees that no starvation will occur.

One problem with this approach is that it has a single point of failure. If the coordinator fails, the system must elect a new coordinator with some appropriate electing algorithm. This new coordinator must poll the other processes to create a new request queue before the processes in the system can enter the critical section again.

### 3.2.2. A token-passing approach

The token-passing approach is another method to provide mutual exclusion to a critical section [SGG02, p600]. A token, that is a special type of message, is circulated around in a logical ring. When a process receives the token, it can enter the critical section by keeping the token. When the process leaves the critical section, it passes the token forward to the next process in the ring. If a process doesn't want to enter the critical section, when it receives the token, it just passes the token forward to its neighbor.

This approach also guarantees mutual exclusion to the critical section since only the process with the token may enter the critical section. If the ring is unidirectional this system also guarantees freedom from starvation.

Two types of failures can occur in a token-passing system. The first failure that may occur is a lost of the token. Then an election must be called to generate a new token. The second failure that may occur is that a process fails. In this case the ring has to be reconstructed with an appropriate algorithm.

## 3.3. Choice of the most appropriate algorithm

This literature study comprises two different kinds of approaches, synchronization of physical clocks and mutual exclusion to a critical section. Now it is time to determine which algorithm that is the most appropriate to use when implementing the betting system. The three algorithms are Cristian's Method, which is a clock synchronization algorithm, the centralized approach and the token-passing approach, which are mutual exclusion algorithms.

The three algorithms are listed in table 3-1, where they are compared to each other according to the parameters in the columns.

| | Fast | Complexity | Expensive | Possibility to cheat | Chosen algorithm |
|---|---|---|---|---|---|
| Cristian's Method | Yes | Low | No | Yes | No |
| Centralized Approach | Yes | Low | No | No | Yes |
| Token-Passing Approach | No | High | Yes | No | No |

**Table 3-1. Comparison of the synchronization algorithms.**

Cristian's method is an algorithm that is fast since the client only has to timestamp the message before it can send the message. This method is also inexpensive since no messages are sent away with short intervals. But using Cristian's method, in the betting system, has a large problem. The client can cheat by time stamping a bet with an incorrect timestamp. In that way it can place a bet after it has expired. Therefore, Cristian's Method is unsuitable to use in this betting system.

The token-passing approach requires that a token have to be passed around in the ring. Because of the token, the algorithm is very expensive, since it has to be passed around in the ring even if all processes are inactive (i.e. doesn't place any bets).

This approach is also very slow and has high complexity since a process must wait for the token to be able to enter the critical section. If the ring is large (i.e. many clients are logged on) and the process that wants to enter the critical section just sent away the token to the next process when it wants to enter the critical section. Then it must wait until the token is sent around the ring before it can enter the critical section (i.e. before it can place a bet).

The centralized approach requires that three messages have to be sent when a client wants to enter the critical section (i.e. wants to place a bet). The Centralized approach is an inexpensive algorithm since nothing is sent when a process is inactive (i.e. doesn't place any bets). It is also fast and has low complexity since only two messages have to be sent before the process can enter the critical section (i.e. place a bet) and one message to notify that is has left the critical section.

Both the centralized approach and the token passing approach prevent the clients from cheating (i.e. place bets after the time limit has expired). But the centralized approach is faster, has lower complexity and is less expensive; therefore it is the most appropriate (of these three) algorithms for this betting system.

# 4. System Design

This chapter describes the system design that was stated in the beginning of the work. Some changes have been made during the implementation. First some design choices and system restrictions are stated. Then the design of every component and the interfaces to the other components is described separately.

## 4.1. Design Choices

### 4.1.1. Programming language

Java was chosen as programming language in the whole system mostly because of its portability. Only some operating system specific settings have to be changed if the system will be moved to another platform.

### 4.1.2. Communication interface

Since Java was chosen as programming language, Java RMI was chosen as communication interface. A short tutorial to Java RMI is presented in appendix A. This tutorial gives readers, with no knowledge of Java RMI, a short introduction. After reading the tutorial it will be easier to read the rest of the report.

### 4.1.3. Database

PostgreSQL was chosen as database engine since it was available at the department. Since Java was chosen as programming language, JDBC was chosen as communication interface between the database server and the database. A short tutorial to PostgreSQL and JDBC is presented in appendix B. This tutorial gives readers, that don't have any knowledge about PostgreSQL and JDBC, a short introduction to these tools. After reading the tutorial it will be easier to read the rest of the report.

## 4.2. System Restrictions

The following restrictions were stated to make the implementation easier:

- Every bet has fix odds.
- The bookmaker has an unlimited amount of money.

## 4.3. Requirement feedback

The rest of this chapter presents the betting system. It describes all components and the communication interfaces on every component. When a requirement has been fulfilled, there will be comment either within parentheses or in the comment field in an interface table. The comment will have the following format:

**"**This fulfills requirement 2.4 a)**"**

## 4.4. System Components

Figure 4-1, below, is an illustration of the components in the system and how they communicate with each other. If the arrow between two components is pointing in both directions it means that remote invocations are performed in both directions. Otherwise remote invocations are only performed in one direction.

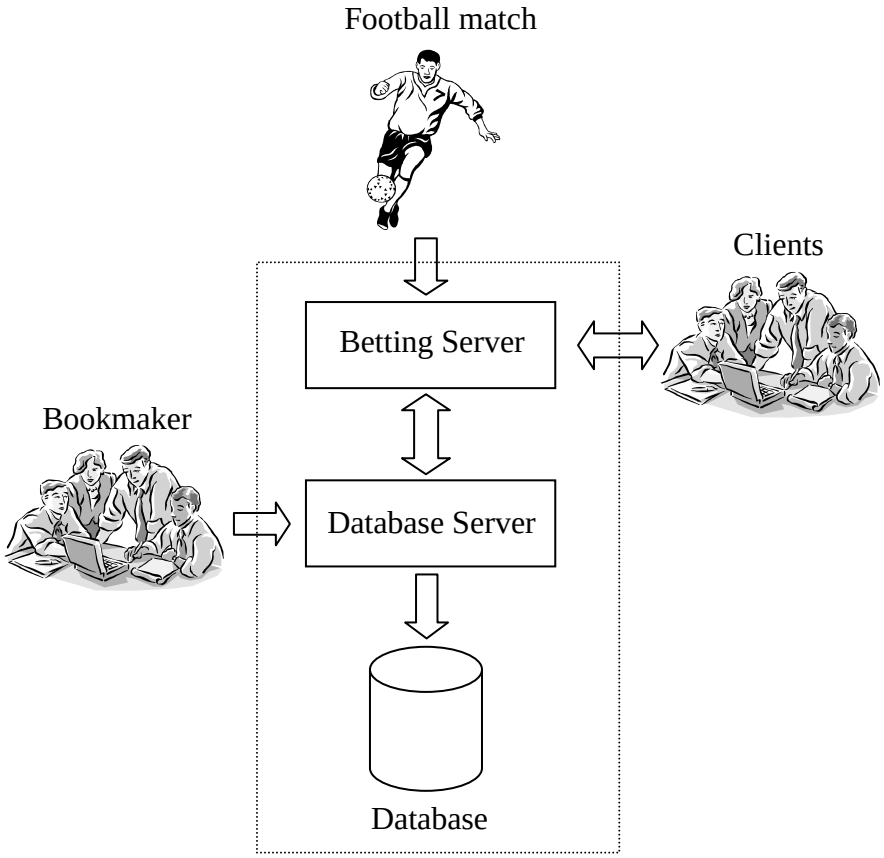


**Figure 4-1. System components in the Betting System.**

The dotted rectangle marks the scope of the betting system. Compare the figure above with figure 2-1 in the requirements. The components outside the dotted rectangle are the actors in the betting system. The components in the betting system communicate, with each other, according to the interfaces described below.

### 4.4.1. Betting Server

The betting server is a Java application that is the central server in the betting system. The betting server communicates with a football match, the clients and the database server. The communication interface between the football match and the betting server is one-directional. The football match component invokes methods on the betting server to simulate a football match. The football match interface is described in table 4-1 below.

| Method | Purpose | Comment |
|---|---|---|
| footballLogin | This method is used when a footballMatch wants to login to the system. It authenticates the user with the database server. | |
| footballLogout | This method is used when a footballMatch wants to logout from the system. It removes information about the login session from the betting server and from the database server. | |
| getTeams | This method is used a footballMatch wants to get information about the teams in a match. | |
| startHalf | This method is used when a half of a football match starts. | |
| endHalf | This method is used when a half of a football match ends. | |
| newBet | This method is used when a new bet shall be generated. | The bet will expire 20 seconds after a newBet invocation. This fulfils one part of requirement 2.6.1 a) and 2.5.1 e). |
| goal | This method is used when a team has scored. | |

**Table 4-1. The FootballMatch Interface.**

The communication interface between the betting server and the clients are bi-directional. This means that both components can invoke methods at each other. The betting server saves information about the clients when they log on to the system (This fulfils requirement 2.5.2 a)). The client interface is described in table 4-2 below. The interface that the betting server can use on the client is described in the client section below.

| Method | Purpose | Comment |
|---|---|---|
| clientLogin | This method is used when a Client wants to login to the system. It autenticates the user with the database server. | This fulfils requirement 2.5.2 a) |
| clientLogout | This method is used when a user wants to logout from the system. It removes information about the login session from the betting server and from the database server. | |
| getUserInfo | This method is used when a client wants to get it's own user information. | This fulfils requirement 2.5.2 c) |
| updateUserInfo | This method is used when a client wants to update it's own user information. | |
| placeBet | This method is used when a client wants to place a bet. | This fulfils one part of requirement 2.5.2 b) |

**Table 4-2. The Client Interface.**

The communication interface between the betting server and the database server is also bi-directional. The betting server invokes methods at the database server to perform operations on the database, e.g. login, user information and football match operations. The database server invokes the saldoUpdate-method, on the betting server, to inform a client that is logged on that the bookmaker has updated the clients account balance. This interface is described in table 4-3 below.

| Method | Purpose | Comment |
|---|---|---|
| saldoUpdate | This method is used when a bookmaker have updated a client's account balance. It is called by the bookmaker part of the DatabaseServer. | |

**Table 4-3. The DatabaseServer Interface.**

The synchronization interface is a separate process on the betting server. When new bets are generated the betting server process informs the synchronization process about the new bet. This process sets a timer that will expire 20 seconds later. The interface contains two methods, *enter* and *release*, which are described in table 4-4 below.

| Method | Purpose | Comment |
|---|---|---|
| enter | This method is used when a client wants to have permission to enter the critical section (i.e. place a bet) | This fulfils requirement 2.6.2 b) one part of requirement 2.5.1 a) and requirement 2.4 a) |
| release | This method is used when the betting wants to give a client permission to place a bet. | This fulfils requirement 2.4 a). |

**Table 4-4. The Synchronization Interface.**

### 4.4.2. Client

The Client component is a Java applet that is the client's interface to the system. This GUI has only some basic functions, such as login, user information and betting operations. It has a background that is a screenshot of the prototype, for viewing football matches live at a cellular phone, which the BASTARD project has developed (this fulfils requirement 2.6.2 a)). The client's interface is described in table 4-5 below.

| Method | Purpose | Comment |
|---|---|---|
| newBet | This method is used when the betting server wants to inform the clients of a new bet. | This fulfils one part of requirement 2.5.2 b) |
| betOutcome | This method is used when the betting server wants to inform a client about the outcome of a bet. | This fulfils one part of requirement 2.5.2 b) |
| saldoUpdate | This method is used when the betting server wants to Inform a client that it's account balance have been updated. | This fulfills requirement 2.5.2 c) |
| reply | This method is used when the synchronization part of the betting server wants to inform a client that it can place a bet (i.e. it can enter the critical section). | This fulfils requirement 2.4 a). |

**Table 4-5. The ClientApplet Interface.**

### 4.4.3. Football Match

This component is a simple Java applet from which a football match can be simulated. The GUI has only a login dialog and some buttons to generate actions that simulate a football match. This component does not contain any interfaces since the communication between football match and the betting server is one-directional.

### 4.4.4. Bookmaker

The BookMaker component is a Java application that administrated the database. The bookmaker is the administrator of the system. The bookmaker accesses the database server through a Java application, which communicates with the database server.

The GUI has a login dialog and three panes that handle different administration objects. The User-pane handles administration of bookmakers and clients. The Match-pane handles administration of matches and the in the BetLog-pane the bookmaker can view placed bets.

The BookMaker component has no interface since the communication between it and the database server is one-directional.

### 4.4.5. Database Server

The Database Server is a Java application that is the link between the system and the database. All access to the database goes through this server. It contains methods to create and update tables in the database. The database server has two interfaces.

The first interface is the bookmaker interface (see table 4-6 on the next page). This interface is the link between the bookmaker and the database. It contains methods to administrate clients, bookmakers and matches. It also contains a method that handles bet logs.

The other interface is the betting interface (see table 4-7 on the next page). This interface is the link between the betting server and the database. It contains methods to handle login and logout. It also contains methods to handle matches and bets.

#### 4.4.5.1 The bookmaker interface

| Method | Purpose | Comment |
|---|---|---|
| login | This method is used when a bookmaker wants to login to the system. It autenticates the user with the database. | This fulfills requirement 2.5.1 a). |
| logout | This method is used when a bookmaker wants to logout from the system. It removes information, about the login session, from the database server. | This fulfills requirement 2.5.1 a). |
| addUser | This method is used when a bookmaker wants to add a new user (bookmaker, client or football match) to the database. | This fulfills requirement 2.5.1 b) and 2.5.1 c). |
| deleteUser | This method is used when a bookmaker wants to delete a user from the database | This fulfills requirement 2.5.1 b) and 2.5.1 c) |
| getUser | This method is used when a bookmaker wants to get user information about a user. | This fulfills requirement 2.5.1 b) and 2.5.1 c) |
| getUserList | This method is used when a bookmaker want to get a list of all client's. | This fulfills requirement 2.5.1 b). |
| getMatchList | This method is used when a bookmaker want to get a list of all matches. | This fulfills requirement 2.5.1 c). |
| updateUser | This method is used when a bookmaker want to update The user information for a user (bookmaker, client or football match) | This fulfills requirement 2.5.1 b) and 2.5.1 c). |
| updateSaldo | This method is used when a bookmaker wants to update for a user. | This fulfills requirement 2.5.1 d). |
| getBetLogList | This method is used when a bookmaker wants to get logs over placed bets. | |

**Table 4-6. The Bookmaker Interface.**

#### 4.4.5.2 The betting server interface

| Method | Purpose | Comment |
|---|---|---|
| login | This method is used when a client or a football match user wants to login to the system. It autenticates the user with the database. | |
| logout | This method is used when a client or a football match User wants to logout from the system. It removes information, about the login session, from the database server. | |
| getTeams | This method is used when a football match user wants to get the name of the teams in the match. | |
| getUserInfo | This method is used when a client wants to get it's own user information. | |
| updateUserInfo | This method is used when a client wants to save it's own user information. | |
| startMatch | This method is used when a football match user wants to start it's match. | |
| stopMatch | This method is used when a football match user wants to start it's match. | |
| startBet | This method is used when a football match user wants to start a new bet. | |
| checkBet | This method is used when a client wants to place a bet. | |
| saveBet | This method is used when a bet expires. | |

**Table 4-7. The Betting Interface.**

### 4.4.6. Database

As mentioned in the design choices, the database engine is a PostgreSQL database. The betting system consists of three different types of users (see figure 4-2 below).

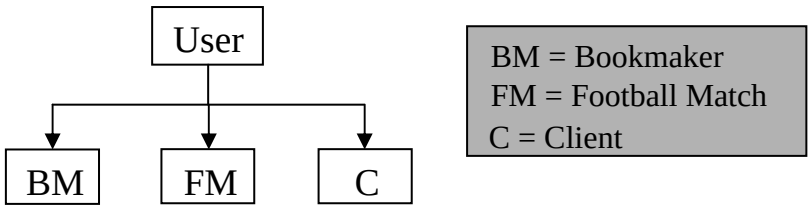


**Figure 4-2. The user hierarchy in the database.**

The Bookmaker is the administrator of the system. The bookmaker logs directly on to the database server to administrate clients and matches. The bookmaker can also get logs of placed bet from the database

A Football Match is a user that simulates a football match. It can only log on to the betting system, generate new bets and new goals.

Clients are users that can place bets. These users can log on to the system, get its user information and perform some basic updates on its user information. Clients can also receive information about new bets and decide whether to place a bet or not.

#### 4.4.6.1 Database tables

The database consists of three different tables. These three tables are:

1) User table
2) AvailableBets table
3) PlacedBets table

The User table contains information about the users in the system (see table 4-8 below). The first three fields are handled equally for all users. The third field indicates the user type. This field determines how the rest of the fields are handled. If the user isn't a client, then the last three fields are ignored since these fields are used to store client specific information.

| Field | Type | Comment |
|---|---|---|
| username | String | The username for this user. |
| password | String | The login password for this user. |
| usertype | Integer | The user type for this user.<br>Zero = Bookmaker (BM).<br>One = Football match (FM).<br>Two = Client (C). |
| firstname / hometeam | String | This field represents the first name of a client (or a bookmaker) or the name of the home team. If the user type field is one, then this field represents the name of the home team. If the user type field is zero or two, then this field represents the first name for a bookmaker or a client. |
| lastname / awayteam | String | This field represents the last name of a client (or a bookmaker) or the name of the away team. If the user type field is one, then this field represents the name of the away team. If the user type field is zero or two, then this field represents the last name for a bookmaker or a client. |
| matchplayed | Integer | Represents a variable that indicates if a match have been played or not.<br>0 = This user is not a match.<br>1 = The match have not been played.<br>2 = The match is<br>3 = The match have been played. |
| cvr | String | A person's civic registration number. If the user is a football match, then this is a empty string. |
| email | String | A person's e-mail address. If the user is a football match, then this is a empty string. |
| accountbalance | Double | This field contains the balance of a users account. This integer is set to –1 if the user is a football match. |

**Table 4-8. The User table in the database.**

The availableBets table contains the bets that the football match users have generated. The fields in the availableBets table are described in table 4-9.

| Field | Type | Comment |
|---|---|---|
| matchusername | String | The username for the user that simulates this match. |
| bet_id | String | The bet id for the this bet. |
| team | Integer | A variable the indicates which team that shall to score within 20 seconds.<br>One = Home team shall score.<br>Two = Away team shall score. |
| odds | Double | This field represents the odds for this bet. |

**Table 4-9. The availableBets table in the database.**

The placedBets table contains the bets that the clients have placed. The placedBets table is described in table 4-10 below.

| Field | Type | Comment |
|---|---|---|
| username | String | The username for this user that placed this bet. |
| matchusername | String | The username for the user that simulates this match. |
| betid | String | The bet id for this bet. |
| amount | Double | This field represents the amount of money that the user choosed to place on the bet. |
| outcome | Integer | This field represents the outcome of the placed bet.<br>1 = The user won this bet.<br>2 = The user lost this bet. |
| time | String | This field represents the time when the bet was placed. |
| hostaddress | String | This field represents the host address where the user place this bet. |

**Table 4-10. The placedBets table in the database.**

# 5. System overview

## 5.1. Class description

This chapter describes the classes in every component of the betting system. It also describes how the classes in a component interact with each other.

### 5.1.1. Skeletons and stubs

The implementation of the betting system contains some skeletons (the class name ends with "_Skel") and some stubs (the class name ends with "_Stub"). The *rmic* compiler generates both classes (see appendix A).

The skeleton class is the server side of the RMI implementation. The skeleton is responsible for dispatching the call to the actual remote object implementation. The skeleton class is also responsible for incoming method invocations. When it receives an incoming method invocation, it reads the parameters for the remote method, invokes the method on the actual implementation and transmits the result back to the caller.

The stub class is the client side of the RMI implementation. It acts as a client's local representation for the remote object. The caller invokes a method on the local stub, which is responsible for carrying out the method call on the remote object. When a stub's method is invocated, the following happens. It initiates a connection with the remote JVM[4] that contains the remote method. Then it transmits the parameters to the remote JVM and wait for the result of the invocation.

[4] Java Virtual Machine

### 5.1.2. Betting Server

This section describes the classes in the betting server and how they interact with each other. Figure 5-1 describes an UML-diagram over how the most important classes and interfaces interact with each other.

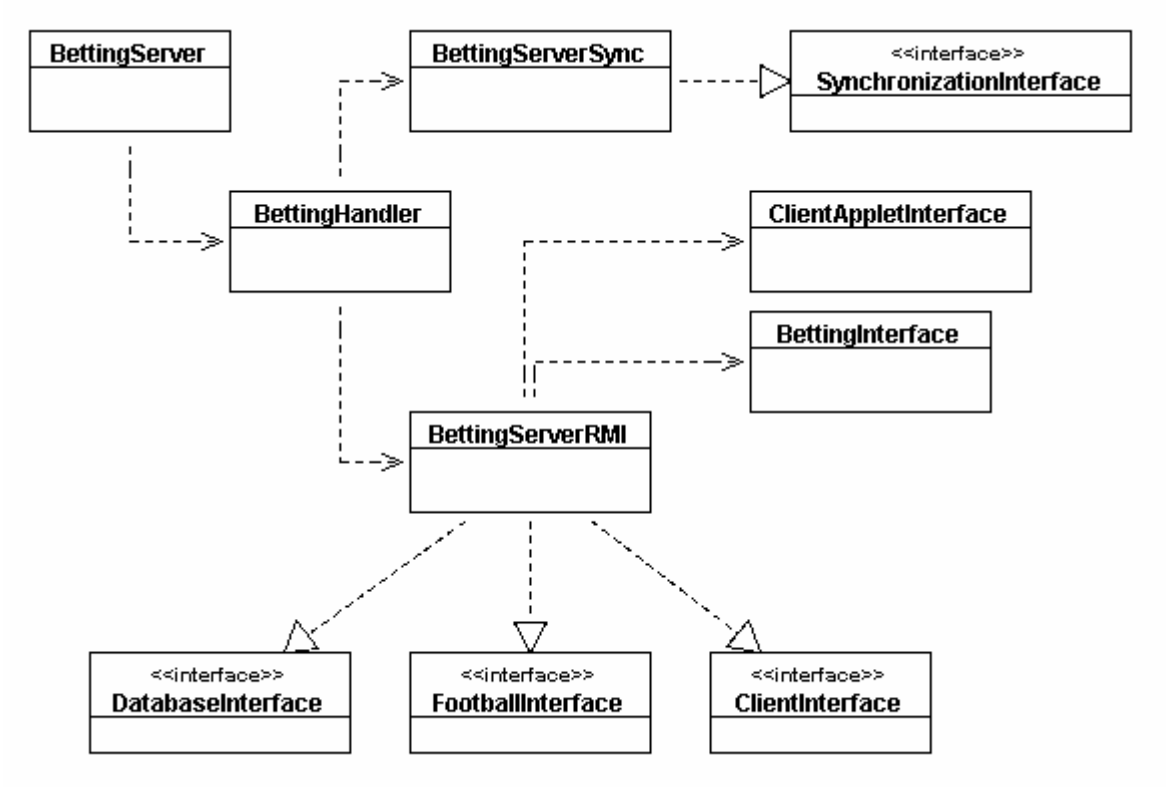


**Figure 5-1. UML-diagram over the most important classes in the betting server.**

**Bet**

This class represents a bet that a client has placed. The objects are created at the Client applet and are passed through the betting server to the database server, where the bet information is stored in the database.

**BetServer_Stub**

This is the stub class for the *BetServer* in the database server.

**BettingHandler**

The class handles the two RMI classes *BettingServerRMI* and *BettingServerSync*. It binds objects of these classes in the RMI registry so the users can find these remote objects.

**BettingInterface**

This class represents the interface, at the database server, that the betting server shall use when it wants to access the database.

**BettingServer**

This is the main class in the betting server. It has only one purpose, which is to create an object of the BettingHandler class.

**BettingServerRMI**

This class contains the remote methods that the other components can invoke at the betting server. It implements ClientInterface, DatabaseInterface and FootballInterface, which are the interfaces the different components can use to access the betting server.

**BettingServerRMI_Skel**

This is the skeleton class for the *BettingServerRMI* implementation in the betting server.

**BettingServerSync**

This class contains the remote methods for the synchronization. It implements the synchronization interface, which is the interface that the client must use when it want to place a bet (i.e. get access to the critical section).

**BettingServerSync_Skel**

This is the skeleton class for the *BettingServerSync* implementation in the betting server.

**ClientAppletInterface**

This class represents the interface that the betting server can access at the client applet.

**ClientInfo**

This class represents a client. It is used when a client wants to get its user information from the database. An object is passed from the database server, through the betting server, to the client applet.

**ClientInterface**

This interface contains the methods that the clients can invoke on the betting server.

**ClientObject**

This is a small class that represents a client’s remote object.

**ClientRMI_Stub**

This is the stub class for the *ClientRMI* implementation in the client applet.

**DatabaseInterface**

This interface contains the methods that the database server can invoke on the betting server.

**FootballInterface**

This interface contains the methods that the football match can invoke on the betting server.

**NewBet**

This class represents a new bet that has been created by the football match component. The object is passed from the football match component, through the betting server, to the client. The new bet is then displayed to the clients, which can decide whether or not it wants to place a bet.

**SynchronizationInterface**

This class contains the interface that is used to handle the synchronization in the betting system.

### 5.1.3. Database Server

This section describes the classes in the database server and how they interact with each other. Figure 5-2 describes an UML-diagram over how the most important classes and interfaces interact with each other.

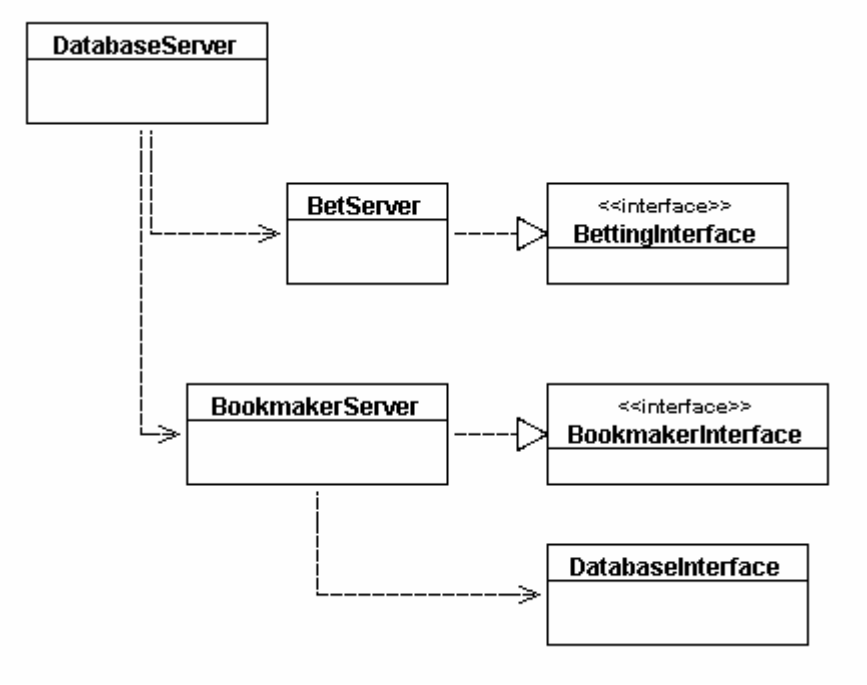


**Figure 5-2. UML-diagram over the most important classes in the database server.**

**BetLogElement**

This class represents an internal class in the BetLogList class. It represents an element in the bet log list.

**BetLogList**

This class represents a list of placed bets that the database server fetches from the database. This list is returned to the bookmaker.

**BetServer**

This class implements the BettingInterface and contains the remote methods that the betting server can use to interact with the database server.

**BetServer_Skel**

This is the skeleton class for the *BetServer* in the database server.

**BettingInterface**

This is the remote interface that the betting server can use to interact with the database server.

**BettingServerRMI_Stub**

This is the stub class for the *BettingServerRMI* implementation in the betting server.

**BookmakerInterface**

This is the remote interface that the bookmaker can use to interact with the database server.

**BookmakerServer**

This class implements the BookmakerInterface and contains the remote methods that the bookmaker can use to interact with the database server.

**BookmakerServer_Skel**

This is the skeleton class for the *BookmakerServer* implementation in the database server.

**DatabaseServer**

This is the main class in the database server. Its only purpose is to bind a BookmakerServer-object and a BetServer-object to the RMI registry so the users can find these objects.

**MatchElement**

This class is a little help class that represents an element in a MatchList-object. It is only used internally in the MatchList-class.

**MatchList**

This class is a list of matches that the database server fetches from the database. This list is returned to the bookmaker.

**UserElement**

This class is a little help class that represents an element in the UserList-object. It is only used internally in the UserList-class.

**UserList**

This class is a list of users that the database server fetches from the database. It is returned to the bookmaker.

### 5.1.4. Bookmaker

This section describes the classes in the bookmaker application and how they interact with each other. Figure 5-3 describes an UML-diagram over how the most important classes and interfaces interact with each other.

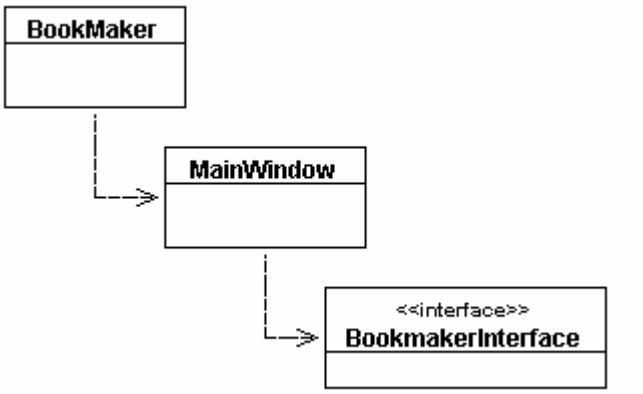


**Figure 5-3. UML-diagram over the most important classes in the bookmaker application.**

**BookMaker**

This is the main class in the bookmaker application. It's only purpose is to initialize the GUI and start the application.

**BookmakerServer_Stub**

This is the stub class for the *ClientRMI* implementation in the betting server.

**LoginDialog**

This class is a login dialog and is used when a user wants to log in to the betting system.

**LoginInfo**

This class contains a username and a password for a user.

**MainWindow**

This is the biggest class in the bookmaker application. It builds up the GUI and handles all interaction between the user and the application.

**User**

This class represents a user. What kind of user it represents depends on the userType varible.

### 5.1.5. Client

This section describes the classes in the client applet and how they interact with each other. Figure 5-4 describes an UML-diagram over how the most important classes and interfaces interact with each other.

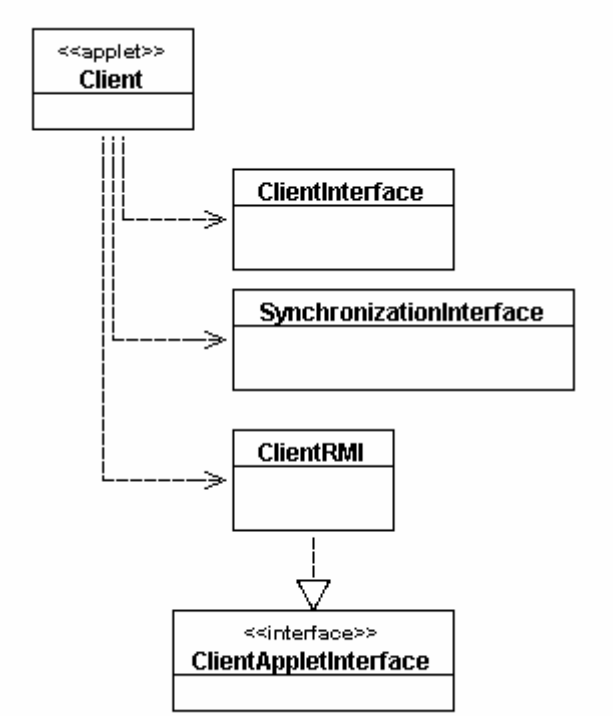


**Figure 5-4. UML-diagram over the most important classes in the client applet.**

**BettingServerRMI_Stub**

This is the stub class for the *BettingServerRMI* implementation in the betting server.

**BettingServerSync_Stub**

This is the stub class for the *BettingServerSync* implementation in the betting server.

**Client**

This is the main class in the client applet. It builds up the GUI of the applet and handles all interaction between the user and the applet.

**ClientAppletInterface**

This is the interface for the methods that the betting server can use to communicate with the client applet.

**ClientRMI**

This class implements the ClientAppletInterface-interface and handles all incomming remote calls from the betting server to the client applet.

**ClientRMI_Skel**

This is the skeleton class for the *ClientRMI* implementation in the client applet.

**LoginDialog**

This class is a login dialog and is used when a user wants to log in to the betting system.

**LoginInfo**

This class contains a username and a password for a user.

### 5.1.6. FootballMatch

This section describes the classes in the football match applet and how they interact with each other. Figure 5-5 describes an UML-diagram over how the most important classes and interfaces interact with each other.

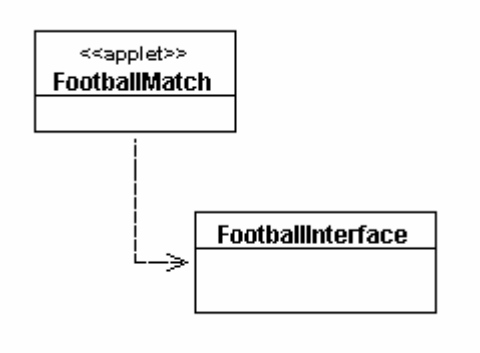


**Figure 5-5. UML-diagram over the most important classes in the football match applet.**

**BettingServerRMI_Stub**

This is the stub class for the *BettingServerRMI* implementation in the betting server.

**FootballMatch**

This is the main class in the football match applet. It builds up the GUI of the applet and handles all interaction between the user and the applet.

**LoginDialog**

This class is a login dialog and is used when a user wants to log in to the betting system.

**LoginInfo**

This class contains a username and a password for a user.

**NewBet**

This class represents a new bet that has been created by the football match component. The object is passed from the football match component, through the betting server, to the client. The new bet is then displayed to the clients, which can decide whether or not it wants to place a bet.

## 5.2. Sequence Diagrams

This section presents some sequence diagrams that describes some typical actions in the betting system. All diagrams are shown in appendix D. The main purpose with this section is to show how the components interact with each other to perform some typical actions

### 5.2.1. The bookmaker edits user information

Figure C-1, in appendix C, is a sequence diagram that shows how remote invocations are performed when the bookmaker logs on to the system and edits user information. All parentheses in this section refer to this figure.

First, the *bookmaker* calls the login-method, on the database server. In the *login*-method, the database server fetches information about the bookmaker from the database server (see *SELECT 1*). The login information is compared with the information that the bookmaker stated.

Then the bookmaker component calls the *getUserList*-method, which gets the list of all users from the in the database (see *SELECT 2*). When the bookmaker has received the list it can choose which user information it wants to edit and then it calls the *getUser*-method which fetches information about the chosen user (see *SELECT 3*).

When the bookmaker has updated the user information it calls the *updateUser*-method to save the updated information. The database server updates the information in the database by performing an update statement on the database (see *UPDATE*). Then the bookmaker calls the logout-method to logout from the database server.

### 5.2.2. The bookmaker receives bet logs

Figure C-2, in appendix C, is a sequence diagram that shows how remote invocations are performed when the bookmaker logs on to the system and fetches bet logs from the database. All parentheses in this section refer to this figure.

First the bookmaker logs on to the system. This procedure is described in the scenario describe above. Then the bookmaker calls the *getBetLog*-method on the database server to fetch the bet log for a user. In the *getBetLog*-method, the database server fetches information, from the placedBets-table, about which bets that the user has placed (see *SELECT 2*).

When the database server knows which bets the user has placed it starts to fetch information about the match, from the user-table, and about the bet from availableBets-table (see *SELECT 3* and *SELECT 4*). The database server fetches information about all bets (see *Bet 1 to Bet 3*) and then it returns a list with information of all bets.

### 5.2.3. A client updates its user information

Figure C-3, in appendix C, is a sequence diagram that shows how remote invocations are performed when a client logs on to the system, fetches it's user information and updates the information. All parentheses in this section refer to this figure.

First the client logs on to the system by calling the *clientLogin*-method, with its username and password as arguments, on the betting server. This method calls the *login*-method, at the database server, which fetches the login information from the database (see *SELECT 1*) and compares it with the information that the client stated. The *clientLogin*-method, at the betting server, stores information that the client is logged on to the system.

When the client has authenticated it self to the betting system, it calls the *getUserInfo*-method on the betting server, which calls the *getUserInfo*-method at the database server. This method fetches the user information (see *SELECT 2*) from the database and returns the information.

When the client has updated the user information, it saves the updated information by calling the *updateUserInfo*-method at the betting server. This method calls the *updateUserInfo*-method at the database server, which updates the information in the database (see *UPDATE*).

When the client wants to logout from the system, it calls the *clientLogout*-method, at the betting server, which removes the login information.

### 5.2.4. A simulation of a football match

Figure C-4, in appendix C, is a sequence diagram that describes how messages are sent when simulating a football match. All parentheses in this section refer to this figure. The diagram assumes that clients are already logged on, to the betting system, when the football match user starts the football match.

The football match user starts the match by calling the *footballLogin*-method at the betting server, which calls the *login*-method at the database server. The *login*-method fetches the login information for the database (see *SELECT 1*) and compares the stored information with the login information that the football match user stated. Then the football match component calls the *getTeams*-method, at the betting server, which calls the *getTeams*-method at the database server. This method fetches team information from the database (see *SELECT 2*) and then the information is returned to the football match component.

When the football match user want to start the match, the *startHalf*-method, at the betting server, is called. The *startHalf*-method calls the *startMatch*-method, at the database server, which checks that the match have not been played (see *SELECT 3*) and then it sets a variable in the user-table which shows that the match is currently being played (see *UPDATE 1*).

In this example, there are no bets or goals in the first half of the match. The next invocation is an invocation to the *endHalf*-method at the betting server.

When the football match user want to start the second half, the *startHalf*-method is called again. In the second half there is one bet and one goal. The rest of this section will show how messages are sent between the components during a bet.

A new bet starts when the football match user presses one of the bet-buttons in the FootballMatch GUI. Then the *newBet*-method, on the betting server, is called. The *newBet*-method calls the startBet-method on the database server, which saves information, about the current bet, in the database (see *INSERT 1*). Then the *newBet*-method informs the synchronization process that a new bet is running. It also informs the synchronization process about the bet key for the current bet. After this it will inform all clients about this new bet by calling the *newBet*-method on every client.

The client places a bet by answering 'yes' in the dialog that is shown when the betting server has called the *newBet*-method. Then the client has to enter the amount that it wants to place on this bet. After this, the *enter*-method, at the synchronization process, will be called. The client will be placed in a queue and when it is empty the synchronization process will call the *reply*-method at the client.

When the client has received a reply-method call it can enter the critical section (i.e. place a bet) by calling the *placeBet*-method on the betting server. The *placeBet*-method calls the *checkBet*-method on the database server. This method will check if the client has enough money to place this bet (see *SELECT 4*) and if so, it will save the bet in the database (see *INSERT 2*).

If the teams scores within 20 seconds (i.e. the football match calls the *goal*-method on the betting server), the client won the bet. Then the betting server will call the *betOutCome*-method at the client to inform the client about the outcome of the bet. After that it will call the *saveBet*-method on the database server to save that outcome of the bet (see *UPDATE 3*) and to update the account balance for the client (see *SELECT 4* and *UPDATE 3*). Then it will inform the synchronization process that the bet has expired.

When the second half is over, the football match user will call the *endHalf*-method at the betting server. This method calls the stopMatch-method on the database server. The *stopMatch*-method updates a variable, which indicates that the match has been played (see *UPDATE 4*). At last, the football match logs out from the betting system by calling the *footballLogout*-method on the betting server.

## 5.3. Screen shots over the GUI's

In Appendix D, some screen shots over the GUI's are presented. Figure D-1 to Figure D4 presented the bookmaker GUI. Figure D-1 presents the User pane of the bookmaker GUI, which is used to administrated clients and bookmakers. Figure D-2 presents the Match pane, which is used to administrate matches. Figure D-3 and Figure D-4 presents the Betting Log pane, which is used to fetch (from the database) and to view the bet logs.

Finally, the last two figures presents the client applet GUI (see figure D-5) and the football match GUI (see Figure D-6).

# 6. Evaluation

The purpose with this evaluation was to determine whether or not this thesis has managed to present the concept behind *rapid betting*. A secondary purpose was to determine if *rapid betting* is an interesting form of betting.

## 6.1. Method

The evaluation was performed through, a short demonstration of the system, for a person. Then the person was asked to fill a short questionnaire. The evaluation questionnaire is presented in appendix E.

The questionnaire can be divided into three parts. First, two questions are asked to determine the person's sports interest. This is important since the *rapid betting* concept depends on that the users, of a rapid betting system, have high knowledge about the sport (in this case football).

Then some questions are asked to determine if the person is interested in betting. This is obviously important since a person probably must be interested in betting to use a *rapid betting* system. Finally, some questions are asked to determine if the person understands the concepts behind rapid betting and if the person would use a system for *rapid betting* (on i.e. a mobile device).

## 6.2. Result

Five persons were involved in this evaluation. These five persons can be divided into three groups. Two persons stated that they are not interested neither in sports nor betting. But according to their answers they understand the concepts behind *rapid betting*.

Two other persons stated that they are very interested in sports but that never or almost never bet. These people also answered that they understand the concept behind rapid betting but they said that they would not use a system for *rapid betting*.

The last person stated that he is very interested in sports and that he use to bet rather often. He stated that he use to bet in his local kiosk and that he never use bet on the Internet. He also understands the concept behind *rapid betting* and he stated that he would probably not use a system for *rapid betting*. This person also gave an interesting comment. He thought that football is an inappropriate sport for rapid betting since it is a rather slow sport with few goals. This is actually a thought that the author has been thinking about.

The conclusion of this short evaluation is that the thesis has managed to present the concepts behind rapid betting. But if someone is thinking about implementing a commercial system for rapid betting, then they have to make a larger investigation to determine if it is profitable.

# 7. Conclusions

This has been a very interesting and instructive thesis for me. I have learned a lot of new things and it has been interesting to build a larger distributed system. I feel that I have managed to create a system that in an easy way presents the idea behind *rapid betting*.

As usual, a lot of problems have turned up during the work. There have been especially two problems that have forced me to put some restrictions on the system. The first problem was that it is not possible, of security reasons, to bind remote object to an *rmiregistry*[5] on another physical computer. Since this was discovered late, this forced me to have an *rmiregistry* on every computer where a client applet is running.

The other problem was that I had problem with the rights to create sockets through a Java applet. I solved this problem by creating a policy file that grants permission, to create sockets, for connecting to processes on other physical computers.

If this prototype shall be further developed to a commercial system, then a lot of things have to be investigated and improved. I will now mention some things that I think must be improved. First and foremost, the problems described above have to be investigated and solved. Those two problems must be solved if the system shall be easy to install and/or be easy move to other platforms.

Another thing that must be investigated are the synchronization algorithms. There are probably a lot of other algorithms (than the three algorithms I investigated) that may be more appropriate in a betting system. To be able to determine which algorithm that is most appropriate a larger investigation has to be performed.

During the evaluation I received an interesting comment from a person. He thought that football is an inappropriate sport for the *rapid betting* since it is a rather slow sport with few goals. As I mentioned in the evaluation, this is actually a thought that has crossed my minds. I personally think that faster sports like basketball, volleyball and handball would be more interesting for *rapid betting*. I think these sports give the users more opportunities to place bets during a match.

Finally I would like to thank my tutor, Greger Wikstrand, who came up with this thesis proposal. He has also been a great tutor and has given many good comments and ideas during the work.

[5] See appendix A for a short description

# 8. References


[CDK01] Coulouris George, Dollimore Jean, Kindberg Tim. Distributed Systems: Concept And Design, Third Edition. Addison Wesley, United Kingdom. 2001.

[NPG03] Neumann, Peter G. Gambling on System Accountability. Communications of the ACM. February 2003 / Vol. 46, No. 2, page 120.

[SGG02] Silberschatz Abraham, Galvin Peter Baer, Gagne Greg. Operating System Concepts, Sixth Edition. John Wiley & Sons, Inc. USA. 2002.

[SUN02] Sun Microsystem, Inc. JAVA™ Remote Method Invocation Specification, Revision 1.8, Java™ 2 SDK, Standard Edition, v1.4, 2002, Downloaded (on 2003-02-24) from the Internet: ftp://ftp.java.sun.com/docs/j2se1.4/rmi-spec-1.4.pdf.

[WJD02] Worsley, John C. And Drake Joshua D. Practical PostgreSQL, First Edition. O'Reilly & Associates Inc., United States of America. 2002.

[WEÖ03] Wikstrand, G, Eriksson, S, and Östberg, F, Designing a football experience for the mobile spectator, The ninth IFIP TC13 international conference on Human-Computer Interaction, International Federation for Information Processing, Laxenburg, Austria, 2003.