Bio - SPHERE

a Service-Oriented Biometrics Platform

Maurits Anton Hartman
Bio - SPHERE
a Service-Oriented Biometrics Platform

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Maurits Anton Hartman
born in Beverly, Massachusetts
United States of America
Voor Marloes
MOVING to Spain at the end of September 2006 was a plunge into the deep on many fronts. I had almost no knowledge of the country and its culture nor the field of biometrics, but it has been in many ways an unforgettable experience. I have discovered many things about Spain, its cities, festivals and culture. On the academic level, I have learned a lot about biometrics and the more general problem of pattern classification and data mining. I managed to overcome some int initial problems and I think the end results is very nice: not only have I created a flexible and fast biometrics platform, I also obtained some good results with the keystroking and multimodal biometrics. My stay in Spain was a perfect mix of culture and science.

I would like to thank the people at the University of Valladolid for all their support, advice and insights they gave. In particular I would like to thank my supervisor, professor Valentín Cardeñoso-Payo as well as Diego, Alejandro, Alberto and Lorena. Your support proved to be invaluable for my project! I would also like to thank my roommates Iván, Jairo, Miguel and Monica, as well as my other friends here for making my stay here truly unforgettable.

A final thanks goes to Marloes and my family for their unrelenting support during the long nine months we were separated, especially during the difficult final weeks.

¡Gracias por todo!

Maurits Hartman
Valladolid, 29th July 2007
Abstract

THIS thesis describes the proceedings of the Bio-SPHERE project. The main goal of this project was to create a flexible and extendable biometrics platform that serves as a tool to do various single and multimodal biometric experiments later in the project. The platform, which is also called Bio-SPHERE, takes the form of a network of cooperating nodes in order to combine biometrics that might be implemented on different operating systems. The biometrics are wrapped inside pluggable modules that can be loaded into the platform at the user’s behest. The biometric functionality contained in the module is made available in the form of services. This service-oriented architecture allows for a great degree of freedom for the users of those services. By creating different compositions of the biometrics services it is possible to perform various multimodal biometrics experiments with different biometric modes using different fusion methods.

Besides the platform, a keystroking biometric was created which was used in the experimentation. Keystroking biometrics is a light-weight biometric that is fairly simple to implement but offers a low biometric performance. Various approaches to keystroking were investigated using a database that is maintained at the University of Valladolid. For the final multimodal experimentation this biometric was fused with a hand geometry biometric in various ways in order to achieve an error rate that is considerably lower than any of the individual biometrics.
# Contents

<table>
<thead>
<tr>
<th>Preface</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xiii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Problem Definition .......................... 3
1.2 Thesis Contributions ....................... 4
1.3 Thesis Organization .......................... 4

## 2 Requirements and Design

2.1 Bio - SPHERE Requirements .................. 7
2.1.1 Project Constraints ....................... 7
2.1.2 Naming Conventions and Definitions .... 9
2.1.3 Functional Requirements .................. 10
2.1.4 Non-Functional Requirements .......... 11
2.1.5 Project Issues ........................... 12
2.2 Bio - SPHERE Architecture ................. 12
2.2.1 Platform Level Architecture .......... 13
2.2.2 Node Level Architecture ............... 14
2.2.3 Interface Design .......................... 16
2.3 Bio - SPHERE Design ........................ 18
2.3.1 Programming Language .................... 18
2.3.2 Operating System Abstraction Layer .... 18
2.3.3 Platform Layer ........................... 19
2.3.4 Core Service Layer ....................... 21
2.3.5 User Service Layer ....................... 22
2.3.6 Client Library ............................ 22
2.3.7 Dynamic Behavior .......................... 23

## 3 Implementation

3.1 Development Platform ....................... 29
3.2 Implementation of the Bio - SPHERE Platform 30
3.2.1 Testing .................................... 31
3.3 Performance Considerations .................. 31
# Multimodal Biometrics 35

4.1 General Characteristics of a Biometric System ........................................... 35
  4.1.1 Performance ......................................................................................... 36
  4.1.2 Multimodal Biometrics .......................................................................... 38

4.2 Feature Level Fusion .................................................................................... 39

4.3 Match Level Fusion ...................................................................................... 39
  4.3.1 Score Normalization ............................................................................... 40
  4.3.2 Fusion Methods .................................................................................... 41

4.4 Decision Level Fusion .................................................................................. 43

4.5 Multimodal Fusion Module .......................................................................... 45

# Keystroking Biometrics 47

5.1 Database ..................................................................................................... 48

5.2 User-Dependent Classification .................................................................... 49
  5.2.1 Preprocessing ........................................................................................ 50
  5.2.2 Simple Approach .................................................................................. 51
  5.2.3 Other Classification Techniques ............................................................ 52
  5.2.4 Results .................................................................................................. 54

5.3 User-Independent Classification .................................................................. 56
  5.3.1 Digraph Approach ............................................................................... 56
  5.3.2 Rhythm-based Approach ..................................................................... 58
  5.3.3 Time-Ranking Approach ...................................................................... 59

5.4 Keystroke Module ......................................................................................... 63

# Multimodal Experiments 67

6.1 Biometric Modes .......................................................................................... 67

6.2 Decision Level Fusion Experiments ............................................................. 69
  6.2.1 Fusion With Two Biometrics ................................................................. 69
  6.2.2 Fusion With Three Biometrics ............................................................... 70

6.3 Match Level Fusion Experiments .................................................................. 72

6.4 Overall Results ............................................................................................. 74

# Conclusions and Recommendations 77

7.1 Conclusions .................................................................................................. 77

7.2 Recommendations ........................................................................................ 78

# Bibliography 79

A Bio-SPHERE Installation Manual 81
  A.0.1 Package Layout ...................................................................................... 81
  A.0.2 Running Bio-SPHERE .......................................................................... 82

B Bio-SPHERE Library Programming 83
  B.1 Introduction .............................................................................................. 83
  B.2 Example Program: Shutdown .................................................................... 85
  B.3 Bio-SPHERE Python Library ................................................................. 88
Contents

C Bio-Sphere Module Programming 89
C.1 The Quote Service 89
  C.1.1 Service Definitions of the Quote Module 89
  C.1.2 Creating the Module Skeleton 90
  C.1.3 The Initialization and Shutdown Functions 92
  C.1.4 The Service Handler Function 92

D Bio-Sphere Graphical Frontend 95

Colophon 97
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Global schedule for the development activities in the Bio-SPHERE project</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>The builtin data types of the Bio-SPHERE platform</td>
<td>22</td>
</tr>
<tr>
<td>5.1</td>
<td>Example keystroke file</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>Results for the user-dependent classification using only printable characters</td>
<td>55</td>
</tr>
<tr>
<td>5.3</td>
<td>Results for the user-dependent classification using all characters</td>
<td>56</td>
</tr>
<tr>
<td>5.4</td>
<td>Results of the digraph-based approach using a Naive Bayes classifier</td>
<td>58</td>
</tr>
<tr>
<td>5.5</td>
<td>Error rates for the various time-ranking methods, applied on different types of keystroke data</td>
<td>62</td>
</tr>
<tr>
<td>6.1</td>
<td>Error rates and computation times for different keystroking classifiers</td>
<td>67</td>
</tr>
<tr>
<td>6.2</td>
<td>Error rates and computation times for different classifiers on hand geometry data</td>
<td>68</td>
</tr>
<tr>
<td>6.3</td>
<td>Results for decision level fusion with two biometrics</td>
<td>69</td>
</tr>
<tr>
<td>6.4</td>
<td>Results of fusing two keystroke classifiers using the Boolean-And rule</td>
<td>70</td>
</tr>
<tr>
<td>6.5</td>
<td>Results of fusing one hand and two keystroke biometrics using majority voting</td>
<td>71</td>
</tr>
<tr>
<td>6.6</td>
<td>Results of fusing one hand and two keystroke biometrics using decision weighting</td>
<td>72</td>
</tr>
<tr>
<td>6.7</td>
<td>Best results of different fusion variants</td>
<td>75</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 General process in a unimodal biometric system. ................................................. 2

2.1 Bio-SPHERE platform overview. ................................................................. 8
2.2 Bio-SPHERE platform architecture. ........................................................... 13
2.3 Bio-SPHERE node architecture. ................................................................. 14
2.4 Subsystems of the Platform Layer. ............................................................... 19
2.5 The major data structures of the Platform Layer. ........................................... 21
2.6 Sequence diagram of the Bio-SPHERE library making a request. ....................... 25
2.7 Sequence diagram of the handling of an incoming service request by the server. ..... 26
2.8 Sequence diagram of the handling of a core service request for the loading of a module. . 27
2.9 Sequence diagram of the handling of a service request that cannot be handled locally. . 28

3.1 Lines of code over time. ................................................................................. 30
3.2 Time parameters of a multimodal biometric request. ........................................ 31
3.3 Time measurements for different request sizes. ............................................. 32

4.1 General process in a unimodal biometric system. ............................................ 36
4.2 Distribution of the genuine and imposter scores. .......................................... 38
4.3 Feature level fusion of two biometric systems. .............................................. 39
4.4 Match level fusion of two biometric systems. ................................................ 40
4.5 Decision level fusion of two biometric systems. ........................................... 43

5.1 Keystroke database structure. ........................................................................ 47
5.2 Different keystroke vectors producing ‘KE’. .................................................. 50
5.3 Keystroke V2 reordered to match V1. ............................................................ 50
5.4 Calculation of latency and duration. ............................................................. 51
5.5 Example training set K. .................................................................................. 52
5.6 Grid-search for lowest error rate using keystroke data. ................................... 59
5.7 Grid-search for lowest error rate using digraph data. ..................................... 59
5.8 ROC curves for the rhythm-based approach using either keystroke- or digraph-based data. The curve is created by varying the threshold while using the missing value penalty for which the lowest error rate is achieved. ................................................................. 60
5.9 Time-ranking process. Vectors T1, T2 and T3 form the training set and are used to compute a template vector which holds the average ranks for the entries A, B, C and D. This template is then used in a rank comparison against a test vector yielding a score. ................................................. 60
5.10 ROC curves for the different time-ranking methods using either keystroke- or digraph-based data. The curve is created by varying the threshold while using the missing value penalty for which the lowest error rate is achieved. ................................................................. 60
5.11 Structure of the keystroke module. ............................................................ 63
5.12 Class diagram of the keystroke companion program. ................................... 65

6.1 Principal hand geometry features. ................................................................ 68
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Distribution of the scores for the keystroking based on the logistic regression classifier.</td>
<td>73</td>
</tr>
<tr>
<td>6.3</td>
<td>Distribution of the scores for the hand geometry based on the naive Bayes classifier.</td>
<td>73</td>
</tr>
<tr>
<td>6.4</td>
<td>Distribution of the scores of the genuine and imposter populations after fusing them with simple sum.</td>
<td>74</td>
</tr>
<tr>
<td>6.5</td>
<td>Distribution of the scores of the genuine and imposter populations after fusing them by taking the minimum sum.</td>
<td>74</td>
</tr>
<tr>
<td>6.6</td>
<td>Distribution of the scores of the genuine and imposter populations after fusing them by taking the maximum sum.</td>
<td>74</td>
</tr>
<tr>
<td>6.7</td>
<td>Distribution of the scores of the genuine and imposter populations after fusing them using matcher weighting.</td>
<td>74</td>
</tr>
<tr>
<td>D.1</td>
<td>Bio-SPHERE Admin's main screen.</td>
<td>96</td>
</tr>
</tbody>
</table>
List of Listings

2.1 Example WSDL definition .......................................................... 16
2.2 XML-based service request and response exchange protocol .................. 17
A.1 Contents of Bio-Sphere source archive ........................................ 81
A.2 Contents of Bio-Sphere installation directory ................................... 82
B.1 Simplified header file biosphere_lib.c ........................................ 84
B.2 WSDL definition of Shutdown service ......................................... 86
B.3 Source code in shutdown.c ......................................................... 86
B.4 Contents of the Python module .................................................... 88
B.5 Example program using the Python module ..................................... 88
C.1 WSDL service definitions for bmod_quote ...................................... 90
C.2 Basic skeleton code in bmod_quote.c ........................................... 91
C.3 The initialization and cleanup functions in bmod_quote.c ................. 92
C.4 The service handler function in bmod_quote.c .............................. 93
Chapter 1

Introduction

With the wide-scale arrival of computers, the concern over their security and privacy has risen as well. Computer users not only face viruses and worms, but also possible identity-theft and loss of information. For companies and governments a high level of computer security is critical. The theft of sensitive company information or an employee who loses his or her laptop can have far-reaching and embarrassing consequences. Global events, such as the increasing threat of terrorism combined with the proliferation of computer systems are catalytic in this trend. A multi-million dollar effort goes into making computer systems more secure through the application of patches and virus scanners. But in most cases the real weakness of the system is the user itself. By using no or weakly selected passwords, using the same password over and over again, or by scribbling passwords on a post-it stuck to the computer, the user itself becomes the weakest link in the security chain. The traditional username and corresponding password paradigm, essentially unchanged since the appearance of the first mainframes in the sixties, also inhibits a strong tradeoff between security and usability: long and random passwords might offer good security but they are cumbersome to use or simply forgotten. Simple passwords or the same password for multiple accounts might improve usability, but offer poor security.

Biometrics can offer a partial solution to this problem by offering superior security over traditional passwords, as well as ease of use. Biometric systems use carefully chosen, unique and unchanging biological properties of people as their ‘passwords’. These traits can be either physiological or behavioral. Physiological traits are biological features that can be observed while the subject is passive, such as fingerprints or an iris pattern. Behavioral features require the subject to be active: such features are for instance a voice sample or the typing rhythm when typing on a computer keyboard. The biometric features have great data density, analogous to a very long password, and tend to be difficult to replay or steal. The permanency of biometric features is also a great advantage of a biometric system; after enrollment in the system, immediate use can be made of it without having to memorize difficult passwords or remembering to update them once in a while. A biometric system only requires a sample to be taken each time which is used in a comparison against a previously stored biometric profile or template. The whole process nowadays only takes a few seconds. Biometrics can not only be used for biometric verification, that is to verify the posed identity of a user using his or her biometrics features, but also for biometric identification. The latter tries to establish the identity of an unknown user by matching the sampled biometric features with those stored in a database. Identification is a more expensive operation since in most cases a comparison has to be made with all the users in the database. Biometric identification is often applied in surveillance contexts, while verification is used mainly for authentication.

Most biometric systems share the same common structure which is depicted in Figure 1.1. The system is comprised of a number of discrete stages, each with a different task. The process begins on the left by taking a sample from the user under consideration. Such a sample can be a picture of the face or a recording of the voice. The sample is then preprocessed to improve its quality or to remove unnecessary data. For instance, in the case of the face picture, the background is clipped leaving only the face. The sample then undergoes the process of feature extraction. In this stage the most essential characteristics are extracted, thereby greatly reducing the amount of data. Features can be for instance the line pattern of a fingerprint or the various dimensions of facial features such as the nose and eyes. These features form the ‘essence’ of the biometric and are stored as the user’s personal profile and is called a biometric template.
During identification or verification the extracted features are compared against an earlier stored template in order to find a match. In the case of verification, an assumed identity is provided alongside the sample and only a single comparison is necessary. In case of identification it is often necessary to compare the features against all users in the database. The pattern matching (also called classification) stage produces a score for the comparison. This score (in case of identification, the highest score) is used by the last stage to base its decision on. Typically the score is compared against a threshold and if it is high enough the system will accept the user: the user has been verified. When doing identification, the user for which the highest score was obtained is returned as the identity.

Although biometrics hold a great promise for security, the technology still copes with a number of problems. First of all, the permanency of the biometric characteristics used by biometric systems raise some serious privacy concerns. While a password can be easily changed, a biometric trait cannot by definition. This prompts many users to distrust biometric systems, fearing a 'Big-Brother' society where their biometric data is used to track their activities and whereabouts. The most suitable features for biometrics, such as the iris pattern of the eyes, tend to be highly intrusive and private. Non-intrusive traits, such as the face, are more readily accepted but offer much lower security in general.

Another major current problem with biometrics is the high error rate resulting in poor reliability (security) and usability. There are some exceptions, but most biometrics require supervised environments to prevent fraud, and databases that are not too large in order to be useful for verification, let alone identification. One possible approach to combat low error rates is to combine (or to fuse as it is called in the jargon) several different biometrics into a single coherent system. This approach, called multimodal biometrics has garnered considerable attention lately and will be investigated in this research.

Biometrics, a decade ago mostly known in research circles, forensics and from Hollywood movies, has become more mature and mainstream in recent years, fueled by an increase in computing power and financial investment. With applications ranging from personal computer use to border control, biometrics are frequently featured as one of the most promising technologies for this century. While there has been made considerable progress, the technology still has to cope with issues such as privacy and low reliability. Multimodal biometrics is seen as a possible solution to these problems since it has the potential to combine nonintrusive, low quality biometrics into a system with a high level of security.
1.1 Problem Definition

Multimodal biometrics, or the combination of various biometrics into a single, coherent system, holds great promise as a way to achieve a lower error rate than the individual biometrics involved. Frameworks for individual biometrics exist, of which the most important one is Bio-API\textsuperscript{1}. This standard is endorsed by a consortium of several companies and is suitable for different biometrics but explicitly offers no support for multimodal biometrics. There is currently no universal software tool for multimodal biometrics application or research. All research is done using either closed-source commercial applications or small ad-hoc programs. These implementations are often specific to the biometrics in question and are not suitable for general use. These programs also tend to cater for only a small number of fusion methods.

A general and flexible tool for multimodal research is highly desirable in order to research different combinations of biometrics and different fusion methods since the development of custom solutions is often time-consuming and costly. In short, such a tool should be able to:

- Easily combine different biometrics into a single system in order to investigate their combined properties in a unified way. Although there is a notion of a common architecture and design to all biometrics, the implementations and operating system vary considerably. The tool should deal with those differences in a flexible way.

- Easily allow for different methods to combine the biometrics, or fuse them. There are different levels and fusion methods to consider and the tool should provide facilities for all of them.

The ECA-SIMM group of the University of Valladolid, in cooperation with four other Spanish universities, is engaged in biometrics research through the BIOSECUR-ID project, funded by the Spanish ministry of Science and Technology. In the framework of this project, a medium sized multimodal biometric database was acquired which includes eight different types (or modes) of biometrics. These modes are face, voice, signature, handwriting, keystroking, fingerprint, iris and palm print. Most of these modes in the database are already actively used in biometric research using algorithms and systems developed at the university. These systems are not general: they are tailored and optimized for a specific biometric mode.

While research for a single biometric can be done using these individual systems, there is currently no straightforward way to conduct research into multimodal biometrics using the multimodal database. Therefore, the University of Valladolid is looking for a way to integrate these individual biometric systems and their corresponding part of the database for the purposes of multimodal experimentation and research. The biometric systems themselves run on a variety of different operating systems and are written in a number of different programming languages. There is often only an ad-hoc design since they are part of active research. A common interface or standard is therefore completely lacking.

The author was invited to the University of Valladolid to work on a project that solves this problem. Since the project is involved with bringing different biometrics together, Bio-SPHERE was chosen as its name.

\textsuperscript{1}http://www.bioapi.org
1.2 Thesis Contributions

The Bio-SPHERE project seeks to address this problem by making the following contributions:

1. **Create an extendable and portable software platform for biometrics.** The platform will also be called Bio-SPHERE and will be used mostly for research purposes at the University of Valladolid, while also keeping options open for use outside of academics. This goal entails the establishment of the requirements, creation of a design and eventual implementation and testing of such a platform.

2. **Create full documentation for this platform.** The Bio-SPHERE platform is to be used by the University of Valladolid and possibly other parties. Complete documentation of the software's internals and interfaces should therefore be delivered to enable future use and extension.

3. **Incorporate keystroking into the platform.** A biometric system based on keystroking will be researched, developed and integrated into the Bio-SPHERE platform to demonstrate the platform’s potential for single-mode biometrics and to do multimodal biometric experiments later on.

4. **Perform multimodal biometric research with this platform.** Several methods of multimodal biometrics will be investigated using the platform and its incorporated biometrics. The biometrics used will be the keystroking biometric developed by the author and a biometric based on hand geometry made available by the University of Valladolid. An important question for the multimodal research is to determine the potential of multimodal biometrics to improve the error rates of low-quality biometrics by combining them.

1.3 Thesis Organization

This thesis is broadly divided in two parts. The first part describes the requirements, architecture, design and implementation of a biometrics platform called Bio-SPHERE and envelops the first two chapters. The second part discusses the various conducted experiments with single and multimodal biometrics, keystroking in particular. A short summary is provided behind each chapter.

- **Chapter 1, Introduction.** This chapter provides an introduction to the thesis and the Bio-SPHERE project itself, outlining its problem definition and goals.

- **Chapter 2, Requirements & Design.** This chapter describes the requirements of the desired software system which will be called Bio-SPHERE. After these are established a proposal for the architecture and the design is presented.

- **Chapter 3, Implementation.** This chapter will follow up on the established architecture and design by describing the implementation of the software platform. Besides implementation of the platform and how it is to be extended, other issues such as the development platform, the testing framework and the platform’s time performance are also discussed.

- **Chapter 4, Multimodal Biometrics.** This chapter will provide a theoretical background on biometrics. The principles of single mode biometrics will be treated to provide the groundwork for a discussion on multimodal biometrics theory.

- **Chapter 5, Keystroking Biometrics.** Besides developing a general biometrics platform, the Bio-SPHERE project also comprises research into keystroking biometrics which will be treated in this chapter. It will provide some theoretic background on keystroking biometrics as well as presenting the results of the various experiments.

- **Chapter 6, Multimodal Experiments.** This chapter will describe the other biometrics used, the various experiments undertaken with multimodal biometrics and their corresponding results, putting the methods set forth in Chapter 4 to practice.
• **Chapter 7, Conclusions and Recommendations.** This chapter will present the results of the Bio-SPHERE project and draw conclusions for both the software engineering effort of the platform as well as the experimentation with biometrics. Several recommendations and proposals for future work are also identified in this chapter.

• **Appendix A, Bio-SPHERE Source Code.** This appendix will describe the Bio-SPHERE software package and how to install it. It will also highlight some parts of the source code.

• **Appendix B, Bio-SPHERE Library Programming.** This appendix will provide a short tutorial on how to use the Bio-SPHERE client library to interface with the platform. This way programs can make use of the various services offered by the platform.

• **Appendix C, Bio-SPHERE Module Programming.** This appendix will provide a short tutorial on how to extend the platform by writing a module for it.

• **Appendix D, Bio-SPHERE Graphical Front-end.** A graphical front-end was created for the platform to ease the testing and experimentation, but also for administration purposes. This appendix describes the features of this program.
Chapter 2

Requirements and Design

This chapter will discuss the requirements, architecture and design of the software platform that is produced in the context of the Bio-SPHERE project. It will first gather the requirements of this platform and then propose an overall architecture for it. This architecture will then be further worked out in a more detailed design. This chapter will provide only the necessary level of detail to gain insight into the inner workings of the Bio-SPHERE platform. This excludes a treatment of all the smaller helper functions involved in the processing. A thorough description of the complete design can be found in the generated documentation online[1].

2.1 Bio-SPHERE Requirements

This section will discuss the requirements of the Bio-SPHERE platform. It starts by describing the constraints imposed on the project in Subsection 2.1.1 and then continues with the functional and non-functional requirements. Finally, the last Subsection 2.1.5 discusses various other issues related to the project. Since Bio-SPHERE is primarily meant to be a research tool, the requirements investigation will not be taken too far.

As stated earlier in the project goals in Section 1.2, the primary purpose of the Bio-SPHERE project is to create an extendable and portable software platform, primarily for multimodal biometrics research. The platform will integrate various biometric modes to this end and provide means to investigate various multimodal fusion methods. The clients and primary benefactors of the project will be the members of the ECA-SIMM research group of the University of Valladolid, while keeping an eye open for (commercial) application of the platform outside the university. The clients of the project, the biometrics researchers, will also be its users. Other stakeholders besides the author, who will be the sole developer, and the clients of the project, will be the developers of the biometric systems for individual modes. They will serve as experts for the particular modes they contribute and will be consulted when integrating their particular biometric.

2.1.1 Project Constraints

This subsection on the project’s constraints will discuss all the restrictions placed on the Bio-SPHERE platform and its development. These constraints place restrictions on the design of Bio-SPHERE and are dictated beforehand.

Solution Constraints The solution offered by the Bio-SPHERE project must provide means to integrate existing biometric technology, either commercial or from the University of Valladolid, into a coherent software system. To this end, the system will offer the individual or combined functionality of these different modes of biometrics through a unified interface to client applications. This interface should be flexible enough to accommodate different biometric modes, technologies and multimodal fusion methods. Details about these biometrics are not available at this point since they are still under active developments and subject to research. What is known however is that the biometrics run on both Linux and Windows so the Bio-SPHERE platform should at least be portable.

to those platforms. It must also be flexible enough to easily integrate these biometrics after the platform itself has been developed. It is also desirable that the platform can run on small and limited embedded devices as well.

**Implementation Environment and Workplace** The Bio-SPHERE platform is part of ongoing biometric research at the University of Valladolid and is initially not intended for non-experimental use. Deployment will therefore be limited to a controlled laboratory environment for experimentation and research purposes. However, it should be possible to extend Bio-SPHERE for real-life application in environments where biometric security is desired. Figure 2.1 illustrates the basic setup of the platform with two biometrics.

![Figure 2.1: Bio-SPHERE platform overview.](image)

Operation typically proceeds as follows. A user, wishing to make use of biometric facilities offered by the platform, issues commands at a front-end application. This front-end serves as the interface between the user and the platform. At the heart of the operations is the Bio-SPHERE core which tries to service the user’s commands using the integrated biometrics and multimodal fusion facilities. The integrated biometric systems retain total control and responsibility over their own operations and will be treated as black boxes. These operations could include the communication with a special hardware device or database. A white-box approach would be impractical since it would pose restrictions on the biometrics that are to be used and require them to be redeveloped in order to adhere to a common interface, which is expensive. In some cases it would also be impossible since commercial systems are only available as closed-source, highly customized packages.

As an example of a typical operation consider a user that wishes to combine or fuse several biometrics contained in the system into a verification operation. He will therefore make a multimodal request using the front-end. The core will then prompt the two biometrics to verify the biometric data provided by the user using the attached hardware devices. The conclusions of the two biometrics are then fused into a single decision and the user is notified. Further research using Bio-SPHERE would possibly involve a batch of test users to be entered into the front-end, using various fusion methods in order to gain insight into the biometric performance of various methods.

**Partner Applications** The Bio-SPHERE platform will provide wrappers around existing biometric technology in order to incorporate them into the platform. Since the biometric technologies predate the Bio-SPHERE platform no common interface or protocol can be enforced on the biometrics for communication with the platform. It is the platform itself that has to be flexible enough to handle most types of biometric systems. The only way to do this is to create a customized wrapper around each biometric mode so that the platform can communicate with the
wrappers and thus the biometrics in a common way. Although there is a common standard for biometrics called Bio-API, it does not allow for multimodal biometrics and is not widely used in research projects.

**Schedule Constraints** The deadline for the total development effort including documentation of the Bio-SPHERE platform will be around the end of June 2007 or early July. At this date, a working beta version shall be produced with a number of integrated biometric modes. This beta release and its included biometric modules shall serve as a demonstration of the platform’s capabilities and to do multimodal experiments. The following Table 2.1 lists the schedule and deadlines for the major activities.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>November 2006 - January 2007</td>
</tr>
<tr>
<td>Architecture</td>
<td>November 2006 - January 2007</td>
</tr>
<tr>
<td>Implementation &amp; Testing</td>
<td>January 2007 - June 2007</td>
</tr>
<tr>
<td>Keystroking Biometrics</td>
<td>May 2007 - June 2007</td>
</tr>
<tr>
<td>Multimodal Experimentation</td>
<td>June 2007</td>
</tr>
</tbody>
</table>

Table 2.1: Global schedule for the development activities in the Bio-SPHERE project.

### 2.1.2 Naming Conventions and Definitions

This subsection will provide a glossary of terms and acronyms commonly used throughout the Bio-SPHERE project.

- **Biometric** A system that can perform verification or identification based on human biological traits. Given a certain user, then the biometric can assert that user’s identity based on this trait. This term can also designate a system which includes only part of the biometric process as will be seen in Chapter [4](#) which is essential for some forms of fusion.

- **Client (Applications)** Software programs that interact with the Bio-SPHERE platform. These could include programs to assist in the experimentation or a authentication program that uses Bio-SPHERE as a backbone. One of the client applications that will be created for the platform is the GUI front-end that is described in Appendix [D](#). For the experimentation several scripts will be created to automate the process and these are also considered client applications.

- **Client Library** The client application do not communicate directly with the platform but instead will communicate with it through a client library. This client library takes away the communication complexity and exposes more simpler primitives to the client applications.

- **Core** The central component of the platform that responds to issued service requests by the users and forwards those to the various biometrics in order to service them.

- **Fusion** The combination of two or more decisions made by different biometrics in order to come up with a single decision. Fusion can be done at different levels using different methods as will be seen in Chapter [4](#).

- **Module** A pluggable software component that adds biometric or other capabilities to the platform. Modules expose the same interface to the platform and the biometrics that will be incorporated into the platform are wrapped inside them.
- **Platform** By platform the complete set of interacting software components including the integrated biometrics is meant; that is, the total set of software components that make up Bio-SPHERE, including interface software for client applications. The terms Bio-SPHERE, Bio-SPHERE platform will also be used to designate platform in this thesis.

- **Request** A data message put forward by a client application indicating a desire for a certain named service offered by the platform.

- **Response** A data message sent out by the platform to a client application in response to a service request made by that client.

- **Service** The platform’s functionality will be thought of as abstract services using a client-server model. A client requests (multimodal) biometrics services to the platform which will forward them to the integrated biometrics which act as the service provider.

**Relevant Facts and Assumptions**

The main assumption that is made with regard to biometrics is that they more or less follow the basic linear process depicted in Figure [4.1](#). That is, they are passive until activated by a request for an verification or an identification of a user and then proceed through a number of steps. In this case it will be the Bio-SPHERE platform who activates the biometric with such a request. Most biometrics can be made to fit this model, but there are some who are event-driven. There are for instance fingerprint systems which are activated when a finger is put on the sensor or a keystroking biometric which monitors users typing on the keyboard in the background. Such systems are difficult to cater for when treating them as a black box and will not be used in this project. None of the systems at the university are of this kind though.

### 2.1.3 Functional Requirements

The Bio-SPHERE platform will possess the following functionality:

**Unimodal Biometric Verification and Identification** The platform must be able to service single mode biometric verification or identification requests coming from a client application. The platform interprets the request, forwards it to the correct biometric module and sends the reply from the module back to the user. It will perform bounds- and error-checking on the request as no assumptions with regard to input tolerance can be made on behalf of the integrated biometrics.

**Multimodal Biometric Verification and Identification** The platform must be able to service multimodal biometric verification or identification requests involving a maximum of ten different biometrics. The provided input data for the various biometrics is checked to be within bounds and error-free and then propagated to the appropriate biometric module. The individual results from the biometrics is then fused using either one of the normalization and fusion methods specified in Chapter [4](#). The fused result is then send back to the user.

**Administration and Monitor Functions** Besides biometric requests, the platform should also provide functions for the shutdown of the platform, the loading and unloading of the biometric modules and to write information to a log file for monitoring and debugging purposes. It should also provide timing mechanisms to assess the time performance of all the operations.
2.1.4 Non-Functional Requirements

This subsection will list the various non-functional requirements of the Bio-SPHERE platform. These requirements can be thought of as attributes or properties to the functional requirements.

**Performance** The time performance of the system is the response time to the various requests made to it. The platform is expected to forward the requests to the appropriate biometrics without adding considerable overhead. In fact the overhead is expected to be in the millisecond range, making sure that the bulk of the response time is taken by the biometrics which typically require a lot of computation. Since the properties of the biometrics that will be incorporated are not yet known, it is not sensible to place restrictions on the allowed response time a priori. The only requirement is that the platforms adds very little overhead to the response time, irrelevant of the platform on which it is running.

Since the platform should also be able to run on very limited platforms computationally-wise, such as embedded systems, its memory footprint should also be very small. Again, no real restrictions can be imposed since the biometrics and their associated databases are expected to have larger memory requirements. Therefore the requirement with regard to memory usage is such that the platform itself uses a fraction of what its biometrics occupy.

**Reliability** The reliability of the platform as a biometric security platform is directly dependent on the incorporated biometrics and the biometric performance. This reliability is defined in terms of several error rates which will be further investigated in the chapter on multimodal biometrics. Since the biometric themselves and their properties are not yet known, no restrictions in terms of minimal error rates will be imposed. The platform itself should be reliable as well by imposing strict bounds- and error-checking on the inputs. Given the research context in which the platform will be used, a failure is not very serious as long as it does not compromise the results of the experimentation. The platform will be subjected to unit and integration testing throughout its implementation, but it is much more likely that the source of failure lies with the biometrics themselves, since they are part of ongoing research and are therefore not expected to be very robust.

**Security** The Bio-SPHERE platform will initially be used in a research context and therefore security is not a primary concern. Its initial application environment will be an isolated laboratory. The multimodal database that is to be used however contains highly personal and sensitive biometric data and is to be used very carefully. The platform itself will only be used during controlled experimentation in the laboratory and it is assumed that the standard security of the operating system provides enough data protection, since the database is already used in that way. If the platform is to be used outside of research contexts then the security restrictions become very stringent, possibly requiring encryption. For this research project however a low level of security will be maintained while keeping options open in the design for future security enhancements.

**Usability** The users of the platform will be mainly researchers and the platform should make it easy for them to extend the platform with their biometrics and to use the platforms functions through an interface library. Both interfaces to the platform should be very well documented and illustrated with examples. The interfaces themselves should be simple and minimal.

**Maintainability** The system is to be used by researchers of the University of Valladolid and possible other (commercial) users. The architecture should therefore be simple and modular to allow for future enhancements such as more security. The platform should be easy to port to other platforms and should be easily adapted to operating system changes. It shall be accompanied by premium documentation and programming guides in order for others to quickly become familiar with the platform’s inner workings. Part of that documentation will be provided in this thesis (see Appendices A, B and C).
Legal The Bio-sphere is a research project, to be licensed under a MIT license which enables others to freely use the program and its source code. For a speedy development of the platform, it is likely that COTS software will be used. If this software’s license prohibits its free distribution as part of the Bio-sphere platform it will not be considered; only (open-source) software whose license is compatible with that of Bio-sphere’s MIT license. Since the MIT license also allows commercial, closed-source adoptions and use of the software, the license of a particular software product must be taken into account when considering it for the Bio-sphere project. Bio-sphere will only provide some example modules for demonstration purposes and the infrastructure of running them. None of the biometrics and their databases will be included in the publicly available package. Legal properties of third-party modules, perhaps featuring patented technology, are outside the scope of Bio-sphere since it only provides a platform for these modules and is therefore the responsibility of the implementor.

2.1.5 Project Issues

A major open issue for the project are the characteristics of the biometrics it is to incorporate. It is known that these are implemented on different operating systems but only guesses can be made about their programming language. Most biometrics are known to have a certain common pipeline architecture as seen earlier but for the rest it is not known how easy it will be to create a suitable wrapper for them. Also the biometric performance properties and stability of the biometrics are not known. This has to be taken into account in the design.

The author has looked for other COTS software but no software has been found that completely fits the bill. Especially the extensibility and required flexibility prompt for a custom design. There exists a standard for biometrics called Bio-API which is backed by a consortium of companies. There is also a reference implementations for this standard but it was found to be unusable after an inspection of its architecture and the blunt fact that it does not support multimodal biometrics at all. Other multimodal biometric systems exist, but offer only two or three modes and are mostly closed-source.

2.2 Bio-sphere Architecture

This section will describe the architecture of the Bio-sphere platform and the motivation for the choices made in this architecture. It will also include a description of the interface design, input and output, and resource management within the platform. Section 2.3 will delve deeper into the issues presented here and present a design to implement them. The platform will be distributed over different computers, will have a service-oriented architecture on the system level and a layered architecture on the node level. The next paragraphs will make a case for these three decisions.

First of all, the Bio-sphere platform will be distributed over various computers and operating systems. Every participating computer in this network will be called a node and contains a local Bio-sphere program that manages the local capabilities (such as biometrics) as well as the connections with the other nodes that make up the platform. The primary reason to distribute the platform over different computers is that the biometrics themselves run on different operating systems and are not easily ported to a single operating system. A face recognition biometric on a Windows machine and a fingerprint biometric on a Linux machine can only be combined if they are wrapped at their local machines and then fused on a platform level. Emulation or virtualization still requires a communication wrapper. The second argument for distribution is the requirement that Bio-sphere should also be available on embedded devices which are often limited in their computational capabilities. Such a device might be capable of taking a fingerprint sample, pre-process it and extract its features, but pattern matching those features against a big database might prove too much for the device, let alone the storage requirements of a database. Distribution of that functionality to more capable machines seems like a natural choice to bring biometrics to small devices. Distribution also allows researchers maximum freedom to work on and develop their biometrics on their operating system of choice. Multimodal fusion research only requires the local inception of the biometric into the platform and not a port to a particular architecture.

The Bio-sphere platform and the nodes that make up the platform will have a service-oriented architecture. All functions of the platform, whether it be those related to biometrics or to multimodal fusion, will be modeled

---

2 Common-off-the-shelf, that is, readily purchasable
as stand-alone services with a unified interface. This architecture makes the platform very flexible and extendable since it allows the user great freedom in combining different services into more complex ones and to extend the platform with new services. Biometrics can be then be viewed as a verification or identification service and multimodal biometrics are just the composition of various individual biometrics into a single multimodal service. A service oriented architecture promotes the separation between the user and the producers of the services, promotes reuse of the services and fits the distributed network topology perfectly. It is also suitable for the heterogeneous mix of operating systems and biometrics that make up the platform, as long as they follow a common protocol to exchange service requests and responses.

On the individual node level, the platform will have a layered architecture, with each layer having a different responsibility. At the bottom, each node will have an Operating System Abstraction Layer so it can be ported to different operating systems. On top of that there will be a Platform Layer is responsible for the servicing of requests, module handling, error checking and maintaining awareness of the other nodes in the network. Some of these functions have a counterpart in the Core Service Layer on top of that. This layer creates wrappers around the platform functions so they can be used as services as well. The top layer or User Service Layer contains the services which can be added dynamically to the node (and thus the platform) by loading modules. The biometric services will reside in this layer. The rationale between this layered approach is to simplify the development effort and to separate responsibility. Layers create a useful abstraction of their underlying complexity so the layer on top of them can focus solely on its own responsibility. Development will be bottom up and after a layer has been implemented and tested, work on the layer on top of it can begin.

### 2.2.1 Platform Level Architecture

As said earlier, the Bio-SPHERE platform will consist of a number of nodes which offer services. Figure 2.2 illustrates the platform with a single client and three nodes each offering a biometric service or the ability to fuse biometrics. Each of the nodes runs on a separate computer and offers the same unified interface for making requests which will be described in Subsection 2.2.3. Therefore for the clients it does not make a difference at which node of the platform they direct their request, although it is probably more efficient computationally-wise to direct it to the local node because less network overhead will be incurred.

![Figure 2.2: Bio-SPHERE platform architecture.](image)

To illustrate the way that this setup services a multimodal biometrics request consider the following example. A client application, running on the same computer as Node 1 makes a request for a multimodal biometric verification using biometrics 1 and 2. The overall multimodal request is a composition of two biometric requests and a fusion request. It makes this request through the client library which forwards it to the local Bio-SPHERE node, Node 1. It is not necessary that the client has further knowledge about the network, only that the Nodes themselves know each other (and especially the services offered by the other). Node 1 services the biometric verification for mode 1 locally and puts out a request for biometric 2 and the fusion to Node 2. Node 1 not necessarily has to know how
Node 2 services those request and that it uses Node 3 for that. Node 2 services the request for biometric 2 and puts out a fusion request (containing the results of the two biometrics) to Node 3. The response of Node 3 is then propagated through Node 2 back to Node 1 and eventually to the client application.

This is just an example of a possible setup of the platform. The architecture is very loose: it only involves a number of nodes that offer services. Therefore it is also possible to have a create a typical client-server situation in which a single node services all the requests made by the other nodes (which run on embedded devices for instance). If the network functionality is not requested it is of course also possible to operate Bio-SPHERE as a single node platform with all the services locally.

### 2.2.2 Node Level Architecture

As explained in the introduction of this section, the nodes that make up the Bio-SPHERE platform have a layered architecture. The four layers, as well as their relations with external entities such as the client application and the rest of the network are depicted in Figure 2.3. The workings of the various layers are illustrated by means of a small example.

On the left a client application can be seen using the interface library to make a request to the node using a network socket provided by the operating system. The request than makes it way up through the OS Abstraction Layer to the platform layer which will forward it to the appropriate service handler. The platform layer consults a registry of available services and decides to forward the request up to the User Service Layer where a biometric module is present which can handle the request.

This module then activates a sensor to take a sample, processes it and compares the biometric features with an earlier stored template from the database. The module then creates a response which it sends back down to the Platform layer. Perhaps the request could only be serviced partly in the node, so the platform then has to look for other nodes than complete the handling. Again it consults its registry and makes a request to a node that can handle the omitting part. This request is then passed down to the OS Abstraction Layer which creates a socket to that node. After the platform layer receives the response from the remote node, it completes the response and sends it back to the client.
This is a just an example to illustrate the events involved in the handling of a service request. Of course a request can be handled only local without external involvement or it can involve one of the services from the Core Service Layer or multiple biometrics. The next paragraphs delve deeper into the four layers.

**Operating System Abstraction Layer**  The sole responsibility of the OS Abstraction Layer is to provide an abstraction of the operating system. That is, provide facilities for the creation and handling of threads, mutexes, network sockets and data files. These facilities are commonly provided through specific libraries or system calls and are highly operating system dependent. Linux and UNIX share some similarity in system and system library calls, but the windows platform is radically different. To make the node impervious to these differences, the OS Abstraction Layer provides a unified interface to the operating system. All the layers on top of it, will be operating system independent.

Ideally the layers on top will only make use of the functions offered by this layer, although some allowances have to be made. The third-party code that is wrapped inside the modules is probably non-portable and quite possibly links to some system libraries and makes use of non-portable system calls. For efficiency reasons some exceptions can be made. For instance, the `printf` function is fairly ubiquitous and creating a special wrapper around it in this layer is overkill.

**Platform Layer**  The platform layer forms the core of the Bio-Sphere node and builds on top of the OS Abstraction Layer. It is responsible for listening for and handling incoming service requests. On startup, a server socket is created by this layer that listens for incoming connections. To handle the requests, it employs a delegation thread model. That is, the main thread spins off the request into a worker thread which will do all the work. The main thread then goes back to listening for more connections.

When a worker thread awakens, it uses the socket it is given to completely haul in the request, decodes it and performs bounds- and error checking on it. If there is any error it will notify the request maker and exit. After the worker thread has checked the connection and request to be safe (which it can do up until a certain height) it consults the registry of the platform layer to look for a suitable handler for the request. After a suitable handler has been found (which might be a remote node), the worker thread forwards the request to it, waits for it to respond, encodes it again for transport and sends it back through its socket. The thread is then discarded.

Besides handling service requests, the platform layers also maintains data structures to keep track of the other nodes in the network, the currently loaded modules and all the available services. It also sets certain options which are passed to the node through the command line at startup or in a configuration file. These options include for instance the maximum number of threads and the port to start listening on.

**Core Service Layer**  The Core Service Layer provides a collection of important platform services. These so-called core services are basically wrappers around certain platform level functions in order for them to be used by clients. Core services include services to load, list and unload Bio-Sphere modules and to print information to the log file. It also contains services to shutdown the node cleanly and to add, list and remove knowledge about other nodes in the platform or to remove specific services. This layer and these wrapper services are necessary since all this low-level functionality is not implemented as a service in the platform layer for efficiency reasons.

**User Service Layer**  The User Service Layer is the topmost layer and contains all the services that are added to Bio-Sphere by its users. These services are implemented in the form of pluggable modules which can be dynamically loaded and unloaded. Each module provides a service handler function which is known by the Platform Layer so it can direct requests to it. The module’s are also aware of the services offered by the rest of the platform so they can build upon those services. Since the code in the module will be executed directly by the platform without inspection or checks, it is the responsibility of the module writers to make sure their modules do not take down the whole node when they encounter an error.
2.2.3 Interface Design

Every node that makes up the Bio-SPHERE platform exposes two interfaces. The first interface is exposed to client applications in order for them to pass service requests to the node. The other interface is exposed by every Bio-SPHERE module. Both interfaces are fairly simple and are explained in this subsection starting with the service request interface. The module interface will be treated superficially in this subsection and more elaborately in the design section.

Service Request Interface This interface will consist of two parts. The first part enables a client application to learn about the services offered by the Bio-SPHERE node, in a way ‘discover’ them. To this end, the platform will use the WSDL 1.1 standard. The other part of the interface is concerned with the exchange of service requests and responses and will use a simple XML-based protocol. The HTTP 1.0 protocol will be the carrier protocol for both types of data. HTTP was chosen both because of its simplicity (it is text-based and stateless), as well as its ubiquity. Most programming languages have HTTP libraries or even built-in HTTP clients. Using HTTP also raises the interesting option to use the Bio-SPHERE platform as a Web-service.

The WSDL or Web Services Description Language is a standard to describe Web services and is seeing use in the context of Service-Oriented Architectures. It describes services and their in- and outputs and seemed quite useful to describe Bio-SPHERE’s services. The standard used for the Bio-SPHERE project is a subset of the 1.1 WSDL standard, which was the latest at the time of development. It is concerned with the definition, not use, of Web services but can also be used to describe services in general. The standard is XML-based and describes the services in a hierarchical fashion. Listing 2.1 provides a very terse example of a WSDL definition.

```
1 <definitions name="ExampleWSDLDefinition">
2   <type name="string"/>
3
4   <message name="ExampleMessage">
5     <part name="partName" type="string"/>
6   </message>
7
8   <portType name="ExamplePortType">
9     <operation name="ExampleOperation">
10        <input message="ExampleMessage"/>
11        <output message="ExampleMessage"/>
12     </operation>
13   </portType>
14
15   <service name="ExampleServices">
16     <port name="ExamplePort" type="ExamplePortType"/>
17   </service>
18 </definitions>
```

List of Listings 2.1: Example WSDL definition

Every WSDL definitions contains a single root element called definitions with an attribute name. In it, it can contain an arbitrary number of type, message, portType and service elements. Type elements describe the data types that are used within the definition. It can be augmented with XML Schema and namespaces but these features will not be supported in favor of more simple parsing. In WSDL, the services are message-based: they accept an input message and return an output or fault message. A message can have multiple parts, each with a specific data type such as a string or integer. The services themselves are defined inside the service elements. Every such element can contain a number of ports, and each port can contain a number of operations. Therefore, there is a three-level hierarchy in the services. The operation elements defines the actual service by specifying
what kind of messages it takes as input and output. A client wishing to discover the services of the node sends an
appropriate command to the service request interface and receives a full WSDL definition of the services which it
can then use to make service requests. It will use an HTTP GET request for this.

The actual transfer of service requests and their subsequent responses will be done using a simple XML-based
protocol on top of HTTP, foregoing the SOAP protocol which is typically used for this but adds unnecessary
complexity and overhead. The protocol is very simple and an example of a service request based on the previous
WSDL definition is given in Listing 2.2. The request itself is sent to the server using a HTTP POST request.

```xml
<request uuid="550e8400-e29b-41d4-a716-446655440000">
  <use service="ExampleService" port="ExamplePort" operation="ExampleOperation" /
  <input message="ExampleMessage">
    <part name="partName" type=""Data data data</part>
  </input>
</request>
```

**List of Listings 2.2: XML-based service request and response exchange protocol**

In this request the use elements specifies the exact service that is requested, while the input element is an
instance of a ExampleMessage. The single data part of that message holds the data string "Data data data",
which in this case can be sent unencoded. In fact most basic data types such as integers, floating point numbers,
dates and strings (with some restrictions) can be sent unencoded. The platform defines some data types for which
encoding is not necessary while user-added data types will be automatically encoded using Base64. The uuid
attribute of the request is a unique identifier based on RFC4122 and the response should match this number when
it is sent back.

**Module Interface** Bio-SPHERE modules will take the form of dynamic loadable libraries, which can be loaded
and unloaded at will by the node. Almost all Operating Systems support the dynamic loading of code and the
OS Abstraction Layer will provide the necessary abstraction functions for it. A module will then define at the
minimum the following data structures:

- **A module information table.** This is a data structure that contains general information about the module
  such as its version and the platform version it expects, as well as a description of the module. Most impor-
tantly, this table contains function pointers to the initialization, cleanup and service handling routines. Upon
  loading, the platform layer will also install a function pointer to the general service handling routine of the
  node in order for the module to use the node’s services.

- **A initialization function.** This function is referred to by the module information table and contains routines
  that should be carried out before module can start servicing. Examples of these could include the opening of
  a data file or connection to a database. The function will be automatically executed to complete the loading
  process and its actual contents are completely user determined.

- **A cleanup function.** This function is called just before the module is unloaded or when the local node is
  shutting down (and thus unloading its modules). It contains cleanup routines to free used memory or close a
  connection to a database for example. It is up to the user to provide a sensible implementation.

- **A service handling function.** This is the function that will be called by the platform layer to have the
  module process a particular request. The real work of the module is performed by this function.
2.3 Bio-SPHERE Design

The design of a Bio-SPHERE node will be discussed in this section. It starts by motivating the choice for the programming language C and then works its way up the layered architecture by covering each layer separately and ending with the client library. The section finishes with a couple of sequence diagrams to illustrate the most important functions.

2.3.1 Programming Language

The programming language of choice to implement the platform is the C language. The choice is primarily motivated by the following three considerations:

- **Portability.** A carefully written program in C is highly portable. C compilers have been around for many years and are available for almost any platform and processor type. By isolating all the operating system dependent code in the OS Abstraction Layer, the platform can be made very portable. Of course the bottom layer remains platform-dependent but a solution has been found to circumvent its non-portability. Subsection 2.3.2 explains the solution in more detail.

- **Interfacing with existing programs and libraries.** Some biometrics come in the form of compiled programs while others are contained in a library. In the first case the C language can make use of system calls (provided by the OS Abstraction Layer) to execute the program and communicate with it using pipes for instance. Or in the case of a library, it can simply be linked to that library. For other programming languages this is less convenient, especially when dealing with compiled libraries. Although projects like SWIG\(^4\) do a pretty good job to create wrappers around libraries, they still require the writing of a considerable amount of glue code for each individual module, as well as other problems.

- **Performance.** The overhead incurred by the processing in the various layers should be very small. Compiled programs are simply much faster than interpreted programs and the overhead added by interpreted languages might not be acceptable on embedded devices.

2.3.2 Operating System Abstraction Layer

This layer is supposed to house all the operating system-dependent code so the layers on top of it can be independent of the operating system. The Apache Portable Runtime library\(^5\) was found to fit the role of this layer perfectly. It is the foundation of the Apache web server and also fulfills an operating system abstracting role for it. The library, and its companion utility library (APR-Util) contain many different facilities such as the handling of threads, network sockets, dynamic library loading, file and directory handling and even XML parsing. Because of its richness in features, its maturity, complete documentation and free availability, the choice was made to go with this library.

A major issue when programming in C is memory managements and the APR library tries to alleviate this problem by making use of memory pools. Memory pools are a dynamic memory allocation technique that require the user to allocate objects not from the heap using calls like `malloc` but rather from a memory pool. A memory pool manages a large chunk of previously allocated memory from which it sub-allocates memory. Instead of having to use the `free` function to de-allocate all the allocated memory, only the memory pool itself has to be deallocated which requires only a single operation. This technique seems like a very powerful memory management technique, especially in combination with the delegation thread model. When a worker thread is created it is also given a fresh memory pool from which it will allocate all its objects. When the thread exits, its pool is destroyed along with it, automatically freeing all possible memory used by the thread.

All the functions of the library that are used by the layers on top of it will no be repeated here since they are very well documented on the APR’s documentation website\(^6\). In subsequent sections references will be made to

\(^4\) [http://www.swig.org](http://www.swig.org)
\(^5\) [http://apr.apache.org](http://apr.apache.org)
\(^6\) [http://apr.apache.org/docs/apr/1.2/](http://apr.apache.org/docs/apr/1.2/)
various APR functions, but their functionality can be guessed from their names, or if not, they will be explained. The library was found to suit all the platform’s needs, including the required XML parsing, making it the only external library used. Therefore, if the APR library can be ported to a certain platform, Bio-SPHERE can be as well.

2.3.3 Platform Layer

The Platform Layer forms the heart of the Bio-SPHERE node and contains a number of subsystems which are depicted in Diagram 2.4. Although strictly speaking the platforms is not object-oriented since it will be written in C, the diagram is very similar to a class diagram since that type of diagram provides useful insight in the properties of the subsystems (which can be thought of as a high-level classes) and their associations.

![Figure 2.4: Subsystems of the Platform Layer.](image)

Subsystems of the Platform Layer

On startup, control is transferred to the Platform Layer which initializes its subsystems through their respective init functions. These functions make sure that the internal data structures of the subsystems are properly set up before the actual servicing of requests starts. After initialization, control transfers to the thread subsystems at which point the system enters into an infinite loop (event_loop) to service incoming requests. When a connection is accepted, the main thread creates a worker thread with setup_thread to handle the request. Every worker thread executes the worker_thread_main function to this end and then exits. The worker threads use one of the HTTP handling functions offered by the HTTP subsystem to send and retrieve data using the socket they are given by the main thread.

The HTTP subsystem is used by the worker threads to handle HTTP requests. It contains the functions handle_http_get and handle_http_post to handle both HTTP GET and HTTP POST requests respec-
tively. The `received_headers` is a utility function that checks if the full HTTP headers have been received; this information is used to determine whether the request is a GET or POST request. These functions will take a socket to extract data from and use functions from both the XML subsystem and Service subsystem to complete their task. The HTTP subsystem also contains two functions to make GET and POST requests itself called `connect_and_get` and `connect_and_post`. These functions will be used to make requests to remote nodes.

The XML subsystem contains various functions to translate between internal data structures and their respective XML representations so they can be transported safely. These data structures, which will be presented in more detail later, represent requests, responses and service definitions. During its translations, the XML subsystem also performs some error- and bounds checking to make sure the data structures are correct. To this end it uses various checks functions offered by the Types subsystem.

The Service subsystem performs the global service handling function which is called `handle_service`. When called, the subsystem consults its service registry to see if the service is available locally. If that is the case, it will forward the request to the appropriate service handling routine. Such a service handling routine could for instance be located inside a module or at the Core Service Layer when it concerns a core service. If the request cannot be serviced locally, then the Service has to look for another node which can. It retrieves service definitions from the various known nodes from the Node subsystem and searches those for a suitable candidate. When found, it will forward the service request using functions from the HTTP subsystem.

The Module subsystem is responsible for the correct loading and unloading of Bio-SPHERE modules in order to extend the range of services offered. It uses dynamic library loading facilities to load the modules and then parses their service definitions (which it propagates to the Service subsystem), calls the module’s initialization routine and installs its service handling functions. When unloading a module, it calls its cleanup routine and removes it from memory. The module also makes sure that only a single copy of a module can be loaded at the time. Modules can be loaded at startup by using the `-m` option with the module’s name.

The Node subsystem maintains awareness of other Bio-SPHERE nodes in the network. It maintains a list of known nodes and the services they offer. Nodes are ordered by their ping time, so the request should be serviced by the nearest node (although this does not have to be the fastest). Nodes can be added by using the `-n` option on startup.

Since the application is multi-threaded there are numerous possibilities for deadlock or race conditions, especially with biometrics inside the modules which are not expected to be thread safe. Therefore the Mutex subsystem exists to provide various mutexes which are used by subsystems to make their operations thread-safe. In particular, this includes the individual service handlers of the modules. The Option subsystem exists to allow the user to set various options for the Bio-SPHERE node, which are passed on the command-line. The various other subsystems query this subsystem for their relevant options, such as the Thread subsystem that checks on which port it should start listening and the maximum number of connections are accepted concurrently.

### Data Structures of the Platform Layer

Diagram 2.5 illustrates the major data structures that are exchanged between the various subsystems in the platform layer, but also in the layers above it. Of particular interest are the `bs_definition`, `bs_service_request` and `bs_service_response` structures. All data structures are prefixed with `bs`, and all of them have a boolean attribute called `from_mp`. This attribute is set to true when the particular data structure has been allocated from a memory pool (and no explicit de-allocation is necessary). If the particular data structure is not allocated from a memory pool it must be explicitly freed. Since the modules allocate these data structures when creating a response and module designers are not obliged to use memory pools (and `malloc` instead, this precaution is necessary.

The data design is designed to follow the structure of the XML standards used with the service definitions, requests and responses. This makes both the parsing from XML to these data structures easier and faster, as well as the translation of the data structures into XML. Of particular interest is the `bs_operation` structure which can contain two or three `bs_message` structures. The minimum requirement is for both an in- and output message, while the fault message if optional. The `bs_data_type` structure is also important and specifies the data type of a particular message part (instance). The type can be built-in in which case it will be type checked during parsing, or custom (by the module’s designer) in which case it will be treated as a sequence of bytes. Type checking is then up to the module. The built-in data types are given in Table 2.3.3.
2.3.4 Core Service Layer

The Core Service Layer makes some of the low-level functionality of a node available as a service. The layer
includes a main service handling functions which will contain a switch for different ports and operations. The fol-
lowing lists the ports and operations contained in the service BiosphereCoreServices. Note that not all subsystems
from the Platform Layer are represented in the Core Service Layer and this because not all services are necessary
for the experimentation. To save time, only a minimum was implemented:

- **ServerAdmin** This port contains the `shutdown` operation which will cleanly shut down the server after a
  specified timeout. This service builds on the `shutdown()` function of the Thread subsystem.

- **ModuleService** This port contains three operations. The `listModules` operation returns a list of the currently
  loaded services. It relies on the `list_modules` function of the Module subsystem. The `loadModule` op-
  erations loads a named module by using `load_module` and the `unloadModule` operation unloads it again
  by calling `unload_module`.

- **NodeService** This port contains two operations. The `addNode` operations adds a node to the list of known
  nodes using `add_node`, while `removeNode` removes it again by calling `remove_node` in the Node sub-
  system.

- **LoggingService** This port contains a single `log` operation that writes a given string to the node’s log file.
### 2.3 Requirements and Design

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bs_bool</td>
<td>A boolean value</td>
</tr>
<tr>
<td>bs_byte</td>
<td>An 8-bit unsigned integer</td>
</tr>
<tr>
<td>bs_int16</td>
<td>A signed 16-bit integer</td>
</tr>
<tr>
<td>bs_uint16</td>
<td>An unsigned 16-bit integer</td>
</tr>
<tr>
<td>bs_int32</td>
<td>A signed 32-bit integer</td>
</tr>
<tr>
<td>bs_uint32</td>
<td>An unsigned 32-bit integer</td>
</tr>
<tr>
<td>bs_int64</td>
<td>A signed 64-bit integer</td>
</tr>
<tr>
<td>bs_uint64</td>
<td>An unsigned 64-bit integer</td>
</tr>
<tr>
<td>bs_double</td>
<td>A double-precision floating point number</td>
</tr>
<tr>
<td>bs_string</td>
<td>A null-terminated string of characters</td>
</tr>
</tbody>
</table>

Table 2.2: The builtin data types of the Bio-SPHERE platform.

#### 2.3.5 User Service Layer

The topmost layer houses the services that are added by the users through modules. As stated before, a module must define a certain data structure so the platform can make use of it. This data structure is called `bs_module` and it must be defined in the module under the name `bsmod_symtable` in order for the Module subsystem to find it. The `bs_module` data structure will contain the following information:

- **Module version information.** The major and minor number of the module.
- **Bio-SPHERE version information.** The Bio-SPHERE version that is expected by the module.
- **General information.** This will include the name, author, date, license and a description about the module.
- **Function pointer to an initialization and cleanup function.**
- **Function pointer to the service handler routine.** This function must have a certain prescribed signature and interprets a given request and creates the appropriate response.
- **Name of the WSDL definition file.** The module is accompanied by a WSDL definition file that describes its services. Upon loading, this file is read and parsed.

For more information about the workings of a module, please consult the module programming guide in Appendix C.

#### 2.3.6 Client Library

The client library, which will provide the service interface to the client applications of Bio-SPHERE, shares a lot of similarity in functionality with the Platform Layer. Besides some glue code, it will only contain a copy of the HTTP subsystem to make HTTP GET and POST request to a Bio-SPHERE node using the `connect_and_get` and `connect_and_post` functions. It will also feature the same XML subsystem for the translation between internal data structures and their XML representations. It exposes only four functions:

- **init_library** The library will also make use of the APR and therefore must be explicitly initialized.
- **cleanup_library** To cleanup the APR and cleanly exit the library.
• `get_services_list` This function will make a HTTP GET request to the specified Bio-SPHERE node to retrieve its service definitions. These are returned in XML form and are transformed into `bs_definition` after reception by the library and then returned to the user.

• `make_service_request` This function will translate the given `bs_service_request` structure into XML and then send it to the node using a HTTP POST. The response is then translated from XML into a `bs_service_response` structure and returned to the user.

An example on how to use the library to make service requests to Bio-SPHERE can be found in the library programming guide in Appendix B.

### 2.3.7 Dynamic Behavior

After having established the static structure of a Bio-SPHERE node, this section will expand on that by specifying the dynamic structure. Especially the mechanisms involved in the handling of service requests which is of course the main function of the platform. This explanation will be done using a number of sequence diagrams for the main operations. Please note that these diagrams are simplifications in some cases in order not to clutter them with all the possible helper functions involved.

The most appropriate place to start is with the Bio-SPHERE library from which all requests originate. The sequence diagram for the issue of a service request by this library is shown in Diagram 2.6. Before any requests can be made though, the library has to be initialized in step 1 by calling `init_library`, which also initializes the APR library (on which also the library is built). Once that is done, the client application can make a request by passing a `bs_service_request` structure to the `make_service_request` function. This function is mostly a front of the `connect_and_post` function of the HTTP Subsystem which does the bulk of the work. The request is first translated into the custom XML format (Listing 2.2) and is then send as a HTTP request to the server by using APR calls for socket creation and sending. The HTTP request is mainly the prefixing of the right headers before the XML. The `receive_response` function is then used to wait for and parse the incoming HTTP response containing the result. The XML response is then translated into a `bs_service_response` structure which is returned to the client application. If at any point there is an error, a specific error code is returned indicating the type of failure. After the client is done with making requests, it closes the library by issuing `cleanup_library` which also terminates the APR.

On the other end, at the Bio-SPHERE node’s side, the handling of an incoming request is depicted in Diagram 2.7. When the server is started, a listening socket is created by the Thread subsystem using functions provided by the APR. When a request does arrive, a separate worker thread is created for it using `apr_thread_create` to create and `apr_pool_create` to give it a private memory pool to allocate from. After the thread is finished, only this pool needs to be destroyed to cleanup all traces of the thread. The worker thread uses functions from the HTTP Subsystem to handle the HTTP POST request. In general, the HTTP headers can be retrieved in the first `apr_socket_recv` so a differentiation between GET and POST can already be made. In the service request handling case, the `handle_http_post` function will then be called and subsequent calls to `apr_socket_recv` make sure the complete XML request is hauled in. This request is then translated into a `bs_service_request` and passed to the main service handler function contained in the Service subsystem. The `handle_service_request` selects the final request handler and will be treated in Diagram 2.8. The returned `bs_service_response` is translated to XML and then sent back over the socket after which the worker thread is destroyed.

Diagram 2.8 illustrates the handling of a core service request for the loading of a module into the Bio-SPHERE node. The `handle_service_request` function examines the request and searches its service registry for a matching service (service name, port name and operation name). It also checks if the input message is of the correct type and contains the correct number and types of parts. It then passes the request to the core service handler. The Core Services Layer’s handler proceeds by calling the correct function in the Platform Layer, in this case the `load_module` function in the Module Subsystem. This function takes the name of a module and tries to load it into memory using the APR’s dynamic shared object facilities. If this succeeds, it looks for the `bs_mod_symtable` data structure which contains pointers to the initialization, cleanup and service handling functions of the module.
as well as the name of its WSDL file. This file is read, parsed into a xml:namespace prefix="ns1" ns1:wsdl:Definition and these definitions are then added to the Service Subsystem. A service response is created indicating whether the loading succeeded. The handling of other types of core services or user services in modules is highly analogous: search for the handler, pass it the request, and return its response. Or in the case of an error, back propagate the error code.

In the case that the node gets a request it cannot service locally, it will consult with its Node Subsystem to see if it can forward the request to a node which can service it. This mechanism is illustrated in Diagram 2.3. Once again the handle_service_request function is passed a request and it searches the registry for a suitable handler. If this search comes up negative it will then call get_servicing_node in the Node Subsystem which return an TCP/IP endpoint if it finds a suitable node. If that is the case, the request is passed to that node using the connect_and_post function in much the same way as the Bio-SPHERE library makes a request.
1. Client app. starts up.
   - init_library()

2. Client app. makes request
   - make_service_request()

3. Request is translated into XML
4. Request is sent to the server
5. HTTP response is parsed and response received
   - apr_socket_create()
   - apr_socket_send()
   - receive_response()
   - apr_socket_recv()
   - apr_socket_close()

6. XML response is parsed into internal data structure and returned to the client app.
   - request_to_xml()
   - response_from_xml()
   - bs_service_response

7. Client shuts down the library
   - cleanup_library()
   - bs_service_response

Figure 2.6: Sequence diagram of the Bio-SPHERE library making a request.
Thread Subsystem

HTTP Subsystem

XML Subsystem

Service Subsystem

APR

1. Initialization of the server

init_listen_socket()

2. Start loop and accept incoming connection

setup_thread()

3. Start new thread and process HTTP POST with the service request.

received_headers()

handle_http_post()

request_from_xml()

bs_service_request

handle_service_request()

response_to_xml()

XML

apr_socket_send(), apr_socket_close(), apr_pool_destroy()

4. Parse XML request into internal data structure

5. Handle the service request

6. Convert the response into XML for transport

7. Send the response, close the socket and exit the worker thread

8. Start waiting again for incoming connection

Figure 2.7: Sequence diagram of the handling of an incoming service request by the server.
1. Initialization of the services subsystem. Read core service WSDL file

2. A core service request comes in, indicating a module load

3. The module (dynamic library) is loaded and the symbol table is looked up in it.

4. The module’s WSDL file is parsed into a definition

5. The module’s services are added to the service subsystem.

6. The module’s initialization routine is called

7. A service response is generated indicating whether the loading succeeded

Figure 2.8: Sequence diagram of the handling of a core service request for the loading of a module.
1. An incoming request is received that cannot be handled locally.

2. The Node Subsystem is queried can be serviced by another node. If so, that node's address is returned.

3. The request is POSTed to that node using the HTTP subsystem.

4. The remote response is returned to the caller.
Chapter 3

Implementation

HAVING established a design for the platform in the previous chapter, this chapter will follow up on that design with the implementation. Section 3.1 will describe the development platform which was used. In fact, BioSPHERE was implemented and tested on two different operating systems at the same time. This section will also detail the tools and programs used during the implementation process. With the development platform established, the discussion then moves to the actual implementation in Section 3.2. The main implementation choices will be explained and motivated, without going into too much detail. Certain crucial parts of the software will be highlighted though, and also the testing framework will be discussed. A treatise of the implementation of the various modules that will be used for the biometric experiments will be deferred until later chapters. This chapter concludes with a number of experiments that were undertaken to gain insight into the time performance of the platform itself (not its modules).

3.1 Development Platform

Operating System  Bio-SPHERE was developed on both a Linux system (Ubuntu Feisty Fawn) as well as a Mac (Mac OS X 10.4). Both machines were equipped with ample processing power and memory. Both operating systems feature a number of common UNIX utilities and GNU programs which were used during development. After most of the development was complete, a successful port to the Windows platform was made using Cygwin. This port was only to investigate the portability of Bio-SPHERE; all further experimentation was done on either Linux or Mac OS X.

Programming Language  The C programming language was chosen as the main implementation programming language and the compiler of choice was gcc, which comes installed by default. On the Mac, the standard gcc version (4.0.1) was used, and on Linux the newer 4.1.2 version. For debugging, the GNU debugger gdb was used. Several tools were created around the Bio-SPHERE platform, in order to test features or to assist in biometric experiments. The most prominent of these is the graphical front-end described in Appendix D. To speed up development, these tools were written in Python (version 2.5) making use of its graphical binding wxPython. Python comes with built-in facilities for XML parsing and HTTP communication, so a Python interface with the platform was quickly created.

As Bio-SPHERE is meant as a biometrics experimentation platform that will be extended further in the future with modules it is very important that it is meticulously documented in order for other to extend it. To this end, not only the high level design is provided in this document, but also full source code documentation. This documentation was generated by the Doxygen tool. Every single function and data structure of the source code was documented using this tool, which also generated dependency diagrams between the various functions. The HTML documentation generated by Doxygen allows to developers to quickly familiarize themselves with the code base and allows convenient traversal of the code through hyperlinks.

http://www.cygwin.com
http://www.stack.nl/~dimitri/doxygen/
GNU Build System  The GNU build system was used extensively to simplify the building, cleaning and distribution process. The autoconf tool was used to create a configuration script which checks for the availability of the prerequisites, in particular the location of the APR library. The configure script also make sure that Bio-SPHERE’s types have the correct sizes. The automake tool was used to create the necessary makefiles to facilitate the building, cleaning and distribution process. The libtool tool was used to manage the correct building, linking and versioning of the libraries used by the platform, including the plugin modules.

Subversion  The project is hosted by Sourceforge (http://biosphere.sf.net) which offers a free Subversion service, a replacement of the older CVS versioning system. Heavy use of Subversion was made in order to keep the work organized, but also to be able to develop at different machines at the same time. A simple svn update could bring the source tree up to date. Figure 3.1 illustrates the accumulation of source code in the repository over time.

Eclipse  The editor of choice was Eclipse which offers an integrated development environment for several different programming languages, including C and Python. It also features excellent integration with Subversion, the GDB debugger and the GNU build system. Eclipse is available for all the major operating systems, which made development on different machines transparent.

3.2 Implementation of the Bio-SPHERE Platform

Upon the start of development, a source tree was created with separate directories for the server, library and modules. To maximize code reuse between the library and the server, another subdirectory was created to hold the common code. This code was bundled into a convenience library and later linked to both the server and the library. In particular, it houses the Types and XML subsystem and a considerable part of the HTTP subsystem. It also features some basic data structures such as a memory pool-based list implementation and other utility functions. A great example of the benefits of this approach is the fact that the code specific to the Bio-SPHERE interface library (libbiosphere) is only 120 lines.

All of the subsystems of the Platform Layer were implemented in a single C file by the same name. Care was taken to make sure that these files exposed only the necessary functions to interoperate, while the rest of the functions was made static. Very limited use of global variables was made, and in each instance these variables had obvious names to identify them as such. Furthermore, a certain style with regard to function signatures was adopted during programming which took the following form:

bs_status function_name(param1, param2, ..., apr_pool *mp);

This style was quite strictly enforced with almost all functions returning a bs_status indicating whether the function succeeded or not. Also since memory pools are used throughout the platform in favor of malloc, most functions required a memory pool to be passed to them. This style mimics the style used throughout the APR library.

After examining a number of HTTP libraries it was decided to provide an own implementation. This was mostly due to the fact that these libraries were unable to work with memory pools or featured only a HTTP client. HTTP 1.0, which is much simpler than HTTP 1.1 since it does not feature persistent connections, was chosen as the protocol. For the XML parsing the Expat parser from the APR utility library was used. Expat cannot be used directly, but the utility library defines suitable wrapper functions around it that also allow for easy traversal of the
parsed elements and their attributes. These functions were used in turn by the XML subsystem to parse the XML into the appropriate Bio-SPHERE data structures.

### 3.2.1 Testing

The main problem of programming in C is the meticulous care that must be taken with memory management. C allows for extraordinary freedom with the use of memory through pointers, but these also tend to introduce memory leaks and especially null pointers. Using memory pools only alleviates this problem, but cannot solve it on its own. Therefore a strict testing regime was used throughout development. Each of the subsystems was unit tested separately and after those were deemed mature enough they were integrated.

A special test subdirectory was maintained with the unit test programs for the different subsystems. Each of those programs would be compiled together with C file of the particular subsystem and then tested. In most cases it was necessary to temporarily remove the static qualifiers from the various helper functions in order to test them separately. Testing the code that uses network functions proved more challenging. On the receiving end, for functions such as \texttt{handle\_http\_get} and \texttt{handle\_http\_post}, simple Python scripts were used that would send certain well- or malformed HTTP requests. On the sending end this method was used as well with scripts. The library could not be relied upon for this testing since it is based on the same code.

The integration testing was also done by compiling small test programs against the server (or library) code that would make and test service request handling and other platform functions. The \texttt{bsmod\_dummy}, which offers a single echo service, was specifically developed to test module loading, service handling and unloading. For the parts that involved network functionality, Python scripts were used to make certain requests or to proxy for a Bio-SPHERE node to which requests are made.

The testing of the various modules developed was done in a similar way as with the subsystems. The module would be linked against a small test program that tests its various helper functions and its service handler. It must be noted that most of the testing effort was reserved for the server and its library. Especially since some modules were developed late in the project, they were not as thoroughly tested as would have been appropriate.

### 3.3 Performance Considerations

In order to gain insight into the performance properties and scalability of the Bio-SPHERE platform, a number of measurements were taken. A number of time parameters were measured the model depicted in Figure 3.2. This model is valid for all service requests that can be handled by, and involve only, a single node, which will be the typical case. In this setup there is a client that makes a service request to the (local) Bio-SPHERE node. The moment the client makes the call to the \texttt{make\_service\_request} of the library is when the timing process starts.

![Figure 3.2: Time parameters of a multimodal biometric request.](image)

Time parameter $t_{c1}$ measures the time between this moment and the moment at which the first bytes of the
request are received at the server. This parameter will include the translation to XML and creation of the HTTP request by the library and the socket setup and network latency. Time parameter \( t_{p1} \) measures the time between the arrival of the first request bytes and the moment that the suitable service handler, which might be the core service handler or a handler in a module, starts its useful work. Decoding of the HTTP request, parsing of the XML service request and the selection of the suitable handler are thus measured by this parameter. The useful work done in the module is denoted by \( t_w \) and will be quite variable since it completely depends on the type of work. The \( t_{p2} \) and \( t_{c2} \) parameters are similar to the \( t_{p1} \) and \( t_{c1} \) parameters respectively, but measure the times when the response is on its way back. The total time taken from the client’s point of view is the sum of these times:

\[
t_{total} = t_{c1} + t_{p1} + t_w + t_{p2} + t_{c2}
\]  

while the total overhead added by the platform is that time minus \( t_w \). The requirement with regard to the time performance, was that the platform adds very little overhead, especially when compared to the computation time involved with biometrics and other services. Therefore the measurements, which are presented next, do not consider the useful work time \( t_w \). The time taken to train classifiers and perform biometric verifications is given alongside the different biometric approaches for keystroking biometrics in Chapter 5 and the other biometric experiments in Chapter 6. Since those systems are mostly treated as block boxes, no attempt will be made to model their time performances or scalability properties.

To obtain the measurements, the Bio-SPHERE server was extended with calls to \texttt{gettimeofday} at the appropriate places. The \texttt{gettimeofday} returns the current system time in seconds and microseconds since the Epoch\(^5\). The measurements were taken using a small test program linked to \texttt{libbiosphere} that made simple EchoRequest to the EchoService offered by the dummy module (\texttt{bsmod\_dummy}\_so). This service does nothing other than copy the input string to the output response and was augmented by including appropriate calls to \texttt{gettimeofday}. The test program repeated the service request 1000 times to improve accuracy, while using different request sizes. Since the local Bio-SPHERE node, that was going to be measured, used a fixed number of services, the execution time was expected to be totally dependent on the request size (the data string that was going to be echoed). All the developed modules were loaded to give a maximum number of services that have to be searched for each request. Since that search for a suitable service handler consists only of very quick string comparisons it is not expected that the number of loaded services has a great influence on the execution time anyway. The results of the measurements are plotted in Figure 3.3.

![Figure 3.3: Time measurements for different request sizes.](image)

As can be seen, the platform’s execution scales more or less linearly with increasing service requests. This is to be expected since the platform makes no interpretation of the data in the requests except for to copy it,
which is a linear operation. The measurements were taken on a Ubuntu Linux system with a 64-bit 1 GHz AMD Athlon processor with 2 gigabyte of RAM. Depicted values are the averages of the measurements. As will be seen later with the biometrics experiments, the various overhead times of the platform are small as compared to the computation time of biometrics which range in the seconds. Also of interest is that the request sizes involved in those experiments will typically be about 500 to 1000 bytes. Data sizes of more than that are not to be expected.
Chapter 4

Multimodal Biometrics

MULTIMODAL biometrics are a straightforward way to improve the error rates of biometrics and this chapter will provide some background theory to show how multimodal biometrics work and how they achieve better results. Multimodal biometrics combine several different and independent biometrics into a single system through a process called fusion. This fusion can be done in different ways and allows the user to bring out the best of each biometric, creating a system that has lower error rates than any of the individual biometrics. Many biometric systems suffer from unacceptable error rates, especially when relying on nonintrusive or weak features. Combining such low-quality systems can be a great and cheap improvement. Because of these advantages, multimodal biometrics garnered quite some attention in the past few years [Ros04] [Jai04] [Ind03] [Sne03].

This chapter will first explain some general characteristics of a biometric system in Section 4.1 in order to set a baseline. It will describe the general structure of a biometric system and properties that give insight into the quality of a biometric system. Then the three different levels at which biometrics can be fused will be explored: feature level, match level or decision level fusion. Along with an explanation of the method itself, a short analysis will be provided which will give some insight into the expected benefits offered by the module. The chapter concludes with the presentation of a multimodal fusion module for the Bio-SPHERE platform, which will be used in the subsequent experimentation in Chapter 6.

4.1 General Characteristics of a Biometric System

Before discussing the various types and methods of multimodal biometrics, the single mode variant is revisited first to introduce the various components and inner workings inhibited by all biometric systems. Figure 4.1 repeats the basic architecture of a biometric system.

As stated in the introductory chapter of this thesis, the process starts by taking a biometric sample from the user under consideration and proceeds to the right through a number of different stages. This sample can be a picture of the user’s face taken with a camera for the purposes of face recognition for example. The sample is then preprocessed to improve the quality and remove unwanted information. Staying with the face recognition example, this step will involve steps to localize the face inside the picture and to correct for lighting conditions. In general, the preprocessing phase aims to drastically reduce the amount of information present in the sample, while highlighting the useful information.

The feature extraction phase aims to extract or ‘measure’ the relevant biometric features. In the case of face recognition, feature extraction involves the establishment of the dimensions of the various facial features such as the length of the nose or length of the ear and the relations between those. The result of this phase is not a biometric sample but rather a set of biometric measurements which is called a feature vector. This vector forms the basis of the user’s biometric template which is stored in a database during enrollment. The template is later used again during verification and identification.

Pattern matching compares the feature vector under consideration with earlier stored biometric templates from the database. The result is a score, in most cases a numeric value in some particular range. The higher the score, the higher the probability that the two vectors came from the same person. When doing biometric verification only the test vector and the earlier stored template of the user are compared, producing a single score. In the case of
biometric identification, the user is obviously unknown and the sample vector has to be matched against the whole database. The result is a vector with scores from all the users. Only biometric verification will be pursued in this chapter and the experimentation, since identification is not an interesting practical variant for biometric security while it adds considerable complexity to the experimental setup.

The score is used to reach a decision about the user under consideration in the last phase of the process, typically using a threshold. In the case of verification the score has to be higher than this threshold for the user to be accepted; a score below the threshold will lead a rejection. For identification, the user template which had the highest score when compared to the test vector is taken as the identity. If the scores are too low then the system concludes that the user is currently not enrolled. Some systems also return a set of the most likely identities instead of a single one. For the experimentation only verification was pursued.

The enrollment phase also makes use of this process but stops after the feature extraction phase by creating a biometric template consisting of the user’s name (and other data) and the feature vector. In some cases the user is asked to supply multiple samples in order to make the system more resilient to inter-person variations. For instance, a human voice sounds quite different in the morning than in the evening and it might be necessary to capture both morning and day samples to ensure all-day use.

In the three basic operations of a biometric system (enrollment, verification and identification) supervision plays an important role. Enrollment should be supervised to ensure that the templates are coupled to the right identity. Without such supervision, an imposter could enroll in the system posing under a false identity. In strict security environments, verification and identification should also be supervised since biometric systems tend to be susceptible to spoofing and replay attacks. For example, an unsupervised security system based on face recognition might be beaten by holding a photograph of a genuine user before the camera’s lens.

### 4.1.1 Performance

In this thesis, the *time performance* of a biometric system is defined in terms of its time and space consumption. For Bio-SPHERE this amounts to its response time to service requests. This type of performance is highly dependent on the particular biometric used since Bio-SPHERE adds very little overhead to the request handling. During the experimentation, the time performance of the various biometrics will be provided. *Biometric performance* is defined in this thesis as the system’s accuracy in making correct verification or identification decisions. The biometric performance is the main contributor to the system’s reliability as a security tool. There are a number of different metrics that express this accuracy of which the most important are listed below.

- **Error Rate.** This is the total number of mistakes made by the system divided by the total number of decisions it has taken. This is the most general metric and should of course be minimized.
4.1 GENERAL CHARACTERISTICS OF A BIOMETRIC SYSTEM

- **False Accept Rate.** The False Accept Rate, abbreviated as FAR, is the probability that an imposter is classified as a genuine user and thus accepted (false positive). In other words, the chance that a user can access the system while posing as somebody else. This corresponds to a Type II Error in statistics. Like the error rate, the FAR should be as low as possible if any security is to be offered. A related metric is the TRR or True Reject Rate, which is $TRR = 1 - FAR$. Since the true probability may be difficult to establish, an estimation of the FAR is used:

$$FAR = \frac{\text{number of false accepts}}{\text{total number of imposter attempts}} \quad (4.1)$$

- **False Reject Rate.** The False Reject Rate, abbreviated as FRR, is the probability that a genuine user is classified as being an imposter and thus rejected (false negative). In other words, the chances that a genuine user is wrongfully rejected. This corresponds to a Type I Error in statistics. If the system is to be usable then this rate should be minimized as well. A related metric is the TRR or True Reject Rate and can be calculated from the FRR: $TAR = 1 - FRR$. Like the true FAR, the FRR may be difficult to determine so in practice an estimation is used:

$$FRR = \frac{\text{number of false rejects}}{\text{total number of genuine user attempts}} \quad (4.2)$$

- **Equal Error Rate.** Both the FAR and FRR can be influenced by changing certain parameters of the system. For instance, by setting the score threshold higher, the FAR will go down but the FRR will go up. There is an inherent tradeoff between the two rates in real-world biometrics. The Equal Error Rate is the rate resulting from a particular parameter setting of the system in which both the FAR and FRR are equal and generally considered to be a good tradeoff between FAR and FRR. The EER is typically used to quickly compare biometric systems with the system with the lowest EER being the better one.

- **Failure to Enroll Rate.** This rate indicates the percentage of people who could not be verified or identified by the system since they either did not enroll in it or their features were inadequately extracted. The Failure to Enroll Rate is not strictly a reliability metric but appears in some contexts where it is used as a property of a live system. It is only included for completion and will not be further considered.

Note that the FAR and FRR are disjoint since they deal only with the imposter and genuine user classes respectively. The error rate depends completely on the FAR and FRR but cannot be derived directly without knowing the sizes of the genuine and imposter populations. For instance, if the system is only tested with genuine users, then there will be only false rejects and true accepts. Thus the FAR has no meaning, and the FRR will be equal to the error rate. On the other hand, if the system is tested with equal numbers of imposters and genuine users, then the error rate will be the average of the FAR and FRR. In general, the error rate can be obtained by computing a weighted average of the FAR and FRR, where the weights depend on the sizes of the genuine and imposter populations:

$$\text{error rate} = \frac{n_{\text{genuine}}}{n_{\text{total}}}FRR + \frac{n_{\text{imposter}}}{n_{\text{total}}}FAR \quad (4.3)$$

where $n_{\text{genuine}}$ and $n_{\text{imposter}}$ are the number of genuine users and imposters respectively and $n_{\text{total}}$ is the total number of users. Reflecting a tendency from the literature, the FAR and FRR will be the main reliability metrics used in this project, since the FMR and FNMR have less meaning on a system level [Dah03]. Note that the FAR and FRR rates will always be estimations using the formulas given earlier since the true FAR and FRR might be impossible to determine exactly due to the empiric nature of biometrics.

As seen in the earlier discussion, there are two different classes of users. These are the genuine users, which should be accepted, and the imposters, which should be rejected. A typical system is of course not aware of the real class of the user it is trying to recognize, but can only give an estimation in the form of a score. The higher the score, the more confident the system is that the user is genuine. An example of a typical distribution of the scores produced by the system for the class of users is given in Figure 4.2. As can be seen from the graph, the imposters

---

1The TRR and TAR are seldom used since the focus is on errors, not on successes.
tend to get lower scores than the genuine users, but there is also considerable overlap in scores (highlighted region). When a threshold is applied to the scores in order to accept or reject a user, this overlap region causes problems. The part of the imposter population whose scores are higher than the threshold (to the right of it) represent the false positives of the system, while the part of the genuine population to the left of the threshold represent the falsely rejected users. It becomes clear that there is no threshold value that obtains perfect classification because of this overlap. Where to set the threshold is up to the user, but typically one strives to have an equal FAR and FRR.

4.1.2 Multimodal Biometrics

Combining biometric systems into a multimodal system is called multimodal fusion or just fusion. There are numerous ways to achieve this fusion through different fusion levels and scenarios [Ros04]. Levels refer to the different places in the biometric process at which the biometrics can be fused. This is done after the feature extraction phases of the systems (feature level), after the scores are determined (match level or score level) or after the individual system have reached a decision (decision level).

Besides different levels, there are also different scenarios for fusion [Ros04] [FA03]. These scenarios basically entail whether a single biometric trait was used (with multiple classifiers) or multiple traits, but also if a single sensor is used (whose data is then shared among multiple classifiers) or multiple sensors. Using different units (for example left and right eye) also leads to different scenarios. During the experimentation different biometric traits will be investigated using different classifiers. The data for each biometric mode is the same for each classifier. Multiple units are not considered.

Multimodal biometrics offer a number of advantages over single mode biometrics by offering better biometric performance. The improved performance is of course dependent on the individual biometric performances, but also

---

2Receiver Operating Characteristic curve.
on the independence between the systems involved. For example, if an attempt is made to fuse several fingerprint classifiers that make use of the same sample the results might be less than expected since the classifiers are likely to fail on the same subset of the data. By using completely different biometrics one can be sure of total independence among them. Using different biometrics makes the system also much harder to spoof since multiple biometrics have to be fooled at the same time. Having more biometrics makes the enrollment, verification and identification more involved. The added cost of another biometric is nowadays less of a problem because of more powerful computers.

4.2 Feature Level Fusion

Feature level fusion is the most complicated version of the three levels and involves the fusion or combination of the feature sets of the different biometrics. An example is the use of video information to assist a face recognition based on two-dimensional imagery [Zha03]. The fusion of two systems is illustrated in Figure 4.3. Note the requirement of a component that fuses the feature vectors as well as the need for a specialized pattern matcher and multimodal database.

![Figure 4.3: Feature level fusion of two biometric systems.](image)

Fusion at the feature level is believed to be the most powerful of the three levels but is difficult to apply [Jai04]. First of all, the feature vectors of the biometrics involved might be completely incompatible or yield a combined vector that is too big for practical use. For instance, combining facial features with fingerprint features makes no sense at all. The fact that most commercial systems are closed also inhibits fusion at this level. Therefore multimodal fusion at the feature level will not be pursued in this project.

4.3 Match Level Fusion

Fusion at the match or score level lacks the power of feature level fusion, but is much easier to implement since it only requires a score from the individual biometrics. It is widely believed to be the best compromise between recognition power and applicability. Whereas feature level fusion might be impossible due to the differing natures of the biometrics, match level fusion can be applied to any biometric that produces a score. Match level fusion also lends itself better to the use of commercial software since some commercial systems tend to offer scores. Figure 4.4 illustrates match level fusion of two biometrics.
Each of the biometrics takes a sample from the user, processes it and produces a score. The scores are normalized and then fused to produce an overall score for the decision maker. Normalization is required because the individual biometric systems might produce scores with completely different ranges. If the first algorithm produces scores in the range $[1, 10]$ while the second biometric has a range of $[10, 100]$ then the overall score will be dominated by the second biometric. Normalization techniques commonly map the scores to the $[0, 1]$ or the symmetric $[-1, 1]$ range, but in this thesis only the $[0, 1]$ is assumed. There are different ways to normalize scores which will be presented in Subsection 4.3.1. Different methods to fuse the scores are given in Subsection 4.3.2 and represent a selection of commonly used techniques [Ind03] [Sne03] [FA03] [Ros03].

### 4.3.1 Score Normalization

In [Ind03] a number of commonly used normalization techniques are used which will be repeated here and used in the fusion experiments. As stated before, all the normalization techniques presented here map the input score to the $[0, 1]$ range.

**Min-Max (MM)** The Min-Max mapping is a linear mapping of the score $s$ to the $[0, 1]$ range based on the minimum and maximum values of the original range. It requires these to be known beforehand. The normalized score is designated by $n$, while the original range of the score is denoted by $S$.

$$n = \frac{s - \min(S)}{\max(S) - \min(S)}$$

(4.4)

**Tanh (TH)** This normalization maps the score according to the following formula:

$$n = \frac{1}{2} \left[ \tanh \left( 0.01 \frac{2 - \text{mean}(S)}{\text{std}(S)} \right) + 1 \right]$$

(4.5)
4.3 MATCH LEVEL FUSION

Two-Quadrics (QQ) The distributions of the scores of the imposters and genuine have some overlap which is highlighted in Figure 4.2. This means that a portion of the imposter and genuine populations have the same score. The Two-Quadrics aims to reduce the influence of this region by increasing the separation between the two distributions. The center of the overlap region is denoted in the graph with \( c \).

\[
 n = \begin{cases} 
 \frac{1}{c}s^2, & s \leq c \\
 c + \sqrt{(1-c)(s-c)}, & s > c 
\end{cases}
\] (4.6)

Quadric-Line-Quadric (QLQ) The Quadric-Line-Quadric works along the same principles as the Quadric-Quadric method and aims to increase the separation between the imposter and genuine populations. Besides using the center \( c \) of the overlap region, also the width \( w \) of the region is used. Figure 4.2 illustrates both parameters. The width of the region is defined as the difference between the smallest genuine score and biggest imposter score. Both parameter are typically provided alongside a biometric system or have to be estimated.

\[
 n = \begin{cases} 
 \frac{1}{(c - \frac{w}{2})^2}s^2, & s \leq (c - \frac{w}{2}) \\
 s, & (c - \frac{w}{2}) < s \leq (c + \frac{w}{2}) \\
 (c + \frac{w}{2}) + \sqrt{(1-c-\frac{w}{2})(s-c-\frac{w}{2})}, & s > (c + \frac{w}{2}) 
\end{cases}
\] (4.7)

4.3.2 Fusion Methods

In \[Sne03\] and \[Ind03\] a number of common match level fusion methods are given which will be reproduced here. The methods in this section are relatively straightforward and easy to implement. More elaborate strategies exist in the literature but will not be pursued in this project because of their complexity. One example of such a strategy is User Weighting which attaches weights to each user based on the wolf-lamb concept \[Dod98\]. After the fused scores have been calculated a decision is made based on whether the overall score higher than some threshold \( L \).

**Simple Sum (SS)** This is the most widely used match level fusion technique. The normalized scores of the individual biometrics are simply summed together to get an overall score. Here \( d_i \) denotes the overall decision for user \( i \), \( M \) is the number of biometrics and \( n^m_i \) is the normalized score of user \( i \) for biometric \( m \).

\[
d_i = \begin{cases} 
 \text{accept,} & \sum_{m=1}^{M} n^m_i \geq L \\
 \text{reject,} & \text{otherwise} 
\end{cases}
\] (4.8)

When trying to predict the error rates of the combined genuine and imposter populations using Simple-Sum, one has to have knowledge about the distributions for each of the involved biometrics. Combining the individual distributions for either the genuine or imposter distributions into two overall distributions might be too involved. This is because one needs to consider everything on a per-user basis: for each user the scores of the individual biometrics need to be calculated and summed to obtain this distribution.

To analyze why Simple-Sum offers an improvement it is sufficient to look at the average imposter and genuine scores after summing. These scores can be obtained by calculating the a priori probabilities of the genuine and imposter populations for the individual biometrics and sum those \[Dah03\]:

\[
 p_{\text{combined}}(s|\text{genuine}) = \sum_{m=1}^{M} p_m(s|\text{genuine}) \\
 p_{\text{combined}}(s|\text{imposter}) = \sum_{m=1}^{M} p_m(s|\text{imposter})
\] (4.9)

where \( p_{\text{combined}} \) is the a priori probability of the multimodal system, \( s \) is the predicted score and \( p_m \) is the a priori probability for biometric \( m \). These probabilities of the individual biometrics can be directly calculated from the
distributions. It becomes clear that the Simple-Sum method increases the domain of the score. For the individual biometric this domain was $[0,1]$, while for the combined system the range is $[0,m]$. This increased range will probably improve the separability of the genuine and imposter populations. Note that the eventual error rates still depend on the chosen threshold and the combined genuine and imposter distributions.

**Min-Score (MI)** The Min-Score method takes the minimum of the scores as the overall score for the decision.

$$d_i = \begin{cases} 
\text{accept,} & s_i = \min(n_{i1}^1, n_{i2}^1, \ldots, n_{iM}^1) \geq L \\
\text{reject,} & \text{otherwise} 
\end{cases}$$

The same analysis as with the Simple-Sum rule can be applied to the Min-Score rule, then the combined a priori probabilities for the genuine and imposter classes become:

$$p_{\text{combined}}(s|\text{genuine}) = \min_{m=1}^M p_m(s|\text{genuine})$$

$$p_{\text{combined}}(s|\text{imposter}) = \min_{m=1}^M p_m(s|\text{imposter})$$

In general it can be expected that the combined probabilities will come from a single biometric because if that biometric produces low scores for imposters, it is also likely to produce low scores for the genuine users (we assume biometrics for which the separability of the classes is low, with a high overlap region). It will therefore mimic the behavior of this biometric. Furthermore, such a multimodal biometric tends to reject users because it favors the lowest score. This is especially nice if the biometrics involves have a large FAR and a low FRR.

**Max-Score (MA)** The same as the Min-Score rule but taking the maximum score.

$$d_i = \begin{cases} 
\text{accept,} & s_i = \max(n_{i1}^1, n_{i2}^1, \ldots, n_{iM}^1) \geq L \\
\text{reject,} & \text{otherwise} 
\end{cases}$$

The analysis is almost the same as for the Min-Score rule, but in this case the system is biased towards acceptance because it takes the highest score. Application of this rule might improve on the individual biometrics if those systems have a low FAR and a high FRR.

**Matcher Weighting (MW)** This scheme applies weights to the individual biometrics’ score before summing them. The weights are based on the quality of the biometric by making use of its EER, denoted as $e^m$. An accurate system has a low EER and thus a bigger weight than an inaccurate system:

$$s_i = \sum_{m=1}^M w^m n_{im}^m$$

where $w^m$ is the weight belonging to system $m$ which is calculated as:

$$w_m = \frac{1}{e^m \left( \sum_{m=1}^M \frac{1}{e^m} \right)}$$

Since $0 \leq e^m \leq 1$, the weights are also in the $[0,1]$ range. Substituting with the constant sum term with $c = \sum_{m=1}^M \frac{1}{e^m}$, the sum of all the weights is 1 as can be seen:

$$\sum_{m=1}^M w^m = \sum_{m=1}^M \frac{1}{e^m \left( \sum_{m=1}^M \frac{1}{e^m} \right)} = \sum_{m=1}^M \frac{1}{e^m c} = \frac{1}{c} \sum_{m=1}^M e^m = \frac{1}{c} c = 1$$
Matcher weighting is similar to Simple-Sum in that it sums the (scaled) scores of the individual biometrics. The overall a priori probabilities of the scores of the genuine and imposter classes can therefore be computed in a similar way by adding the weights:

\[
p_{\text{combined}}(s|\text{genuine}) = \sum_{m=1}^{M} w_{m} p_{m}(s|\text{genuine})
\]

\[
p_{\text{combined}}(s|\text{imposter}) = \sum_{m=1}^{M} w_{m} p_{m}(s|\text{imposter})
\]

where \(w_{m}\) is the weight assigned to the individual biometric. The matcher weighting has the same benefits as the Simple-Sum rule by creating more separation between the genuine and imposter distributions. It will probably outperform the Simple-Sum because it favors the best biometrics which already feature considerable separation (since their FAR and FRR is low).

### 4.4 Decision Level Fusion

Decision level fusion, as illustrated in Figure 4.5, is based solely on the decisions made by the individual biometrics. For verification this amounts to combining the various accepts and rejects and for identification it involves choosing the most likely identity from the set of identities produced by the biometrics. It is the most simplest method to implement and can be used with any biometric system, including closed-source commercial ones. It also the least powerful of all the three approaches since it is based on such limited information.

Several different methods for decision level fusion can be found in the literature from which a limited selection will be reproduced here [Zue96][Ho94]. Please note that the methods apply to biometric verification, but can be extended to be used for identification.

**Boolean-And (BA)** All the individual biometrics have to accept a user in order to achieve an overall acceptance. Here \(d_{i}\) denotes the overall decision for user \(i\), while \(d_{im}^{n}\) denotes an individual decision for user \(i\) and biometric \(m\). The accept and reject decisions will be interpreted as the Boolean values true and false respectively. The overall
FAR rate will be lower or at most equal to any of the individual FAR rates, but the FRR will be higher. This scheme can be extended for identification by requiring all the proposed identities to be the same as the overall result.

\[ d_i = \bigwedge_{m=1}^{M} d_{mi} \]  

(4.16)

This rule only accepts a user if all the underlying biometrics also accept him or her. Therefore the system only accepts an imposter if all the biometrics accept the imposter. The combined FAR can be estimated by:

\[ \text{FAR}_{\text{boolean, and}} = \prod_{m=1}^{M} \text{FAR}_m \]  

(4.17)

Which will be lower than any of the individual FAR rates. As for the False Reject Rate, this will increase since a rejection by at least one the biometrics will lead to an overall rejection. In general the FRR of fusing \( m \) biometrics with Boolean-And can be computed using the inclusion-exclusion principle:

\[ \text{FRR}_{\text{boolean, and}} = \sum_{m=1}^{M} \text{FRR}_m - \sum_{i,j:1 \leq i < j \leq n} \text{FRR}_i \cdot \text{FRR}_j + \sum_{i,j,k:1 \leq i < j < k \leq n} \text{FRR}_i \cdot \text{FRR}_j \cdot \text{FRR}_k - \ldots - \text{FRR}_1 \cdots \text{FRR}_m \]  

(4.18)

The combined FRR will be equal or larger than any of the individual FRRs. The resulting error rate can be computed using Equation 4.3. The method is a bit crude and the FRR will deteriorate quickly if the number of biometrics increases.

**Boolean-Or (BO)** Similar to the Boolean-And rule, but in this case the OR-operation is used on all the scores, which effectively means that at least one the subsystems should issue an accept for an overall acceptance. This will increase the population coverage and the resulting FRR will lower or at most equal to any of the individual FRRs. The overall FAR will be equal or higher though.

\[ d_i = \bigvee_{m=1}^{M} d_{mi} \]  

(4.19)

By using Boolean-Or the user will be accepted when at least one accept decision is issued. The combined FAR rate can be computed in the same way as the combined FRR was computed for the Boolean-And rule. The combined FRR is the product of the individual FRR rates. Boolean-Or will lead to very low false rejection rates, but increases the false accepts at the same time.

**Majority Voting (MV)** In this scheme all the decisions made by the individual biometrics as a 'vote' in a voting process [Zue96]. If the number of accepting votes for a user is more than some required majority \( L \) than the user is accepted and rejected otherwise. For identification the method is analogous: the biometrics propose an identity and the one with the most votes wins (possibly requiring meeting some threshold). Let \( D_i \) be the set of all Boolean decisions of user \( i: D_i = \{ d_1^i, d_2^i, \ldots, d_m^i \} \) and \( A_i \) a subset of \( D_i \) such that:

\[ A_i = \{ a | a \in D_i, a = \text{accept} \} \]

The decision for user \( i \) will be calculated as follows:

\[ d_i = \begin{cases} \text{accept,} & |A_i| \geq L \\ \text{reject,} & \text{otherwise} \end{cases} \]  

(4.20)
Depending on the chosen threshold, the overall FAR or FRR of Majority Voting can be lower than the original rates. Note that the Boolean-And and Boolean-Or are special cases of Majority Voting with thresholds of \( L = M \) and \( L = 1 \) respectively. Majority Voting requires at least three biometrics to be different than these two methods. The combined FAR rate can be estimated by using the inclusion-exclusion principle. Take for instance the case \( M = 3 \) and \( L = 2 \). Then the FAR rate will be \( \text{FAR}_1 \cdot \text{FAR}_2 + \text{FAR}_1 \cdot \text{FAR}_3 + \text{FAR}_2 \cdot \text{FAR}_3 - \text{FAR}_1 \cdot \text{FAR}_2 \cdot \text{FAR}_3 \).

This can be generalized as follows:

\[
\text{FAR}_{mv} = \sum_{k_1, \ldots, k_L: 1 \leq k_1 < \ldots < k_L \leq M} \text{FAR}_{k_1} \cdots \text{FAR}_{k_L} - \sum_{k_1, \ldots, k_{L+1}: 1 \leq k_1 < \ldots < k_{L+1} \leq M} \text{FAR}_{k_1} \cdots \text{FAR}_{k_{L+1}} + \ldots + \pm \text{FAR}_1 \cdots \text{FAR}_M
\]

(4.21)

The computation of the FRR follows a similar reasoning, except that it requires at least a majority of \( M - L + 1 \) biometrics to issue a false reject in name of the combined system. For \( M \) biometrics the FRR is expressed as:

\[
\text{FRR}_{mv} = \sum_{k_1, \ldots, k_{M-L+1}: 1 \leq k_1 < \ldots < k_{M-L+1} \leq M} \text{FRR}_{k_1} \cdots \text{FRR}_{k_{M-L+1}} - \sum_{k_1, \ldots, k_{M-L+2}: 1 \leq k_1 < \ldots < k_{M-L+2} \leq M} \text{FRR}_{k_1} \cdots \text{FRR}_{k_{M-L+2}} + \ldots + \pm \text{FRR}_1 \cdots \text{FRR}_M
\]

(4.22)

**Decision-Weighting (DW)** This method is very similar to the Matcher Weighting scheme presented in Section 4.3. The calculation of the weights is based on the EER of the biometrics and uses the same formula. An accept decision is represented here as having a value of 1, while a reject decision has a value of 0. After applying the weights and summing them, the decision depends on whether the sum meets a threshold \( L \):

\[
d_i = \begin{cases} 
\text{accept}, & \sum_{m=1}^{M} w^m d_i^m \geq L \\
\text{reject}, & \text{otherwise}
\end{cases}
\]

(4.23)

When analyzing the expected FAR and FRR rates of this method it should be noted that in the case of an imposter, each biometric contributes \( \text{FAR}_m \cdot w^m \) to the overall score on average. The expected score for imposter then becomes:

\[
\text{false accept score} = \sum_{m=1}^{M} w^m \text{FAR}_m
\]

(4.24)

If the threshold is set substantially lower than this value, it is very likely that the system will exhibit a lot of false accepts. The same goes for the false rejects except that the threshold should be higher than the expected score:

\[
\text{false reject score} = \sum_{m=1}^{M} w^m \text{FRR}_m
\]

(4.25)

### 4.5 Multimodal Fusion Module

In order to carry out multimodal biometric experiments using the Bio-SPHERE platform, a multimodal fusion module was developed. The module will incorporate many of the fusion and normalization techniques described earlier in this chapter and will support both match level as well as decision level fusion. The module is designed
to be used in conjunction with biometric services and only expects those services to produce a score (match level) or a decision (decision level). The module is limited to at most three different biometrics which will suffice for the experimentation in this project, but this number can be readily expanded.

The modules offers a single service called BSModFusionServices which contains two ports: MatchLevel and DecisionLevel. These ports contain the methods for the match level fusion and the decision level fusion respectively. The module’s implementation is fairly straightforward with a helper function for each different method. The main service handler dispatches the requests to the appropriate helper function. The requested method is specified by giving its abbreviation, along with optional parameters. The module was designed to fuse two or three biometrics since it was not likely that more biometrics were going to be used simultaneously. The module can be easily expanded to handle more biometrics though. The module is written completely in C and is stateless. That is, no actual initialization or cleanup is necessary.

Fusing two or three biometrics at the match level can be accomplished with the fuse2 and fuse3 operations. These input message to these operations take the following parts:

- **method.** The desired fusion method. This can be "SS" for Simple-Sum, "MI" for Min-Score, "MA" for Max-Score or "MW w1 w2 w3?" for matcher weighting. In the case of matcher weighting, the error rates of the biometrics that are being fused have to be specified. The fusion module transforms these into the appropriate weights.

- **norm1, norm2.** The desired normalizations for the scores of the individual biometrics. 'norm1' specifies the normalization for the score of the first biometric while ‘norm2’ specifies the score of the second one ('norm3 is applied in the case of three biometrics). The accepted values for each of the norms are "NO" for no normalization, "MM min max" for Min-Max normalization, "TH mean stdev" for TanH, "QQ c" for Two-Quadrics and "QLQ c w" for Quadric-Line-Quadric. The parameters for the normalizations have to be provided after their codes. In the case of three biometrics, a third normalization 'norm3' must be provided for the third biometric.

- **score1, score2.** The scores of the individual biometrics to be fused. These scores must be numbers. the fusion module is able to recognize scientific notation as well (eg. 3.0e-34). In the case of three biometrics, a third score 'score3' must be provided.

The responses to the match level fusion requests come in the form of a single fused score which is a number (bs_double). Fusing two or three biometrics at the match level can be done by using the fuse2_decision and fuse3_decision respectively. Since no normalization is used in decision level fusion the input message to these services is simpler:

- **method.** The fusion method to use. Choice are "BA" for Boolean-And, "BO" for Boolean-Or, "MV L" for Majority Voting and "DW L w1 w2 w3?" for Decision Weighting. Majority Voting and Decision Weighting both take an extra threshold while the latter also requires the error rates of the biometrics.

- **decision1, decision2.** The individual decisions of the biometrics that are being fused. The module interprets strings as 'True', 'Accept', 'Accepted', '1' all as being an accept decision. The same goes for reject decisions. In the case of three biometrics, a third decision 'decision3' must be provided.

The responses to the decision level fusion requests contains a single message part which is either 'ACCEPT' or 'REJECT' (bs_string).
Chapter 5

Keystroking Biometrics

Keystroke biometrics is based on the unique way that people type on a keyboard. It is a behavioral biometric that makes use of the different timing characteristics of keystrokes that exist between individual users. Recognizing people by their typing patterns has been done since the 19th century telegraph operators could recognize each other by means of their unique individual rhythms when sending messages over the line. The same went later for radio operators using morse code. Research done during the 1980s showed that typists could be recognized by their typing patterns as well.

The main applications of keystroking biometrics are in the context of biometric verification, especially in combination with passwords. Since users type their passwords almost daily, the keystroke data for them tends to be highly stable and usable for keystroking biometrics. Keystroking biometrics are also used in identification contexts, primarily for surveillance. By constantly monitoring keyboard activity with a keystroke biometric, some certainty can be gained as to whether a genuine user is still typing after he or she logged in. This monitoring amounts to keystroke logging though, and is sometimes restricted or even prohibited by laws, such as the U.S. Patriot Act.

The underpinnings of keystroking biometrics are the timing aspects of pressing and releasing keys. Several timing measurements can be derived from a person’s typing such as the latency (time between successive down presses), duration (the total time a certain key is pressed) and the overall typing speed. Other approaches also consider the applied pressure on the key but this requires a special keyboard. Frequency of errors, left or right handedness, the way capital letters are formed and the use of non-standard keys such as the numpad keys are sometimes also used. Most keystroking biometrics are based on latency and duration of keystrokes.

The main advantage of keystroking biometrics is the ubiquity of suitable hardware since most computers are equipped with a keyboard. Samples can be easily obtained and require little or no preprocessing since the essential information (keystroke and time) can be obtained from the operating system directly. The biometric is also non-intrusive, since it makes use of the keystrokes a user would type anyway. This non-intrusiveness makes the biometric highly acceptable in contrast with other biometrics. Keystroking biometrics also have low computational and storage requirements, making it suitable for application in limited environments and distributed environments since network overhead will also be low. There are even proposals to equip mobile phones with this biometric [Cla07].

A drawback of keystroking biometrics, as with most behavioral biometrics, is that its biometric performance (FAR and FRR) is relatively low compared to biometrics based on physical traits. The high error rate is due to the large inherent variability in keystroke data which is dependent on the physical situation of the user. When a user is tired or relaxed, he or she will exhibit quite a different pattern than when under considerable stress. Also, the collectability of sufficiently large training data sets poses problems. Some users might be unwilling to type a given text of 2000
words during enrollment, preferably during different sessions.

Keystroking biometrics can be divided into text-dependent and text-independent varieties. In the first variety the system makes use of more or less fixed-length keystroke sequences which are known beforehand. A text-dependent keystroking systems is a natural choice to strengthen a password-based authentication system. Text-independent approaches work with arbitrary keystrokes entered by the user. These approaches are more suitable for surveillance applications where constant monitoring of a user’s input is required. Text-dependent systems tend to offer better reliability since they can be more accurately trained, while text-independent systems are more flexible.

As part of the Bio-SPHERE project a keystroke biometric will be created and incorporated into the platform using a database developed by the University of Valladolid. The biometric is meant to demonstrate the single-mode biometric capabilities of the platform and will later on be used for multimodal experimentation. The choice for keystroking was made since it is a relative straightforward and easy to use biometric, requiring no additional hardware. To keep things simple, only biometric verification is considered, although most of the methods presented in this chapter could easily be expanded for identification. After the database is discussed a number of approaches will be investigated, the best of which are implemented into a Bio-SPHERE keystroke module.

In the following discussion a keystroke is defined to be the total action of pressing and releasing a particular key. Thus a keystroke is comprised of two separate events: a key down event of pressing the button and an key up event of releasing it. Operating systems allow for the separate capture of both these events, although only the down-event leads to the production of a character on the screen. The next section will present the database that was used in the research. Next, several user-dependent approaches to the keystroking problem are investigated in Section 5.2. User-dependent in this context means that a separate classification model is trained for each user individually as opposed to a single model for all users. The latter approach is called user-independent and is treated in Section 5.3. The keystroke module for the Bio-SPHERE platform is treated in Section 5.4.

5.1 Database

The Computer Science faculty of the University of Valladolid maintains a moderately sized multimodal database with data sets from students both from the University of Valladolid as well as from the University of Bilbao. One of the modalities of the database is keystroking and this part of the database was used for the keystroking experimentation. The database holds keystroking data of around 130 individuals. For each individual, four separate sessions were recorded at different times and in each session the user was asked to type his or her full name four times for a total of 16 keystroke vectors. In addition to this, the users were also asked to type the names of 12 proceeding users in order to capture imposter data. So in all, the database provides 16 genuine and 12 imposter keystroke vectors for each user. All the keystroke vectors, both genuine and imposter, for a given user are guaranteed to produce that same string of printable characters. This also implies that the vectors can differ by their non-printable characters such as shifts.

The database is file system based and its organization is illustrated in Figure 5.1. For each user, a directory is present with his or her number as a name. Inside each of these directories four more directories, numbered 0001 to 0004 are present reflecting the four different sessions. Each of those directories holds the genuine user data, which is typed by the user him or herself and the imposter files produced by the user. Genuine files are prefixed with the letter g, while the imposter files are prefixed with the letter i. The imposter files produced by a user refer to proceeding users, depending on the session directory and the number of the imposter file. For instance, imposter file i01.txt inside session directory 0002 of user 3005 imposes keystroke data of user 3001 (3005 − 1 − (0002 − 1)³ = 3001). This scheme is cyclic, so the imposter files from the user with the lowest number refer to the twelve users with the highest number as can be seen in Figure 5.1.

The data files themselves are simple text files containing one header line with the number of keystrokes in the file and one line for each individual keystroke event. For each keystroke both the key down and key up events are recorded along with the time difference with respect to the previous event. The first few events of such a file are illustrated in Table 5.1. Events are recorded by their scancode; the mapping to a corresponding key depends on the keyboard and operating system. Note that it is not necessary to know the exact key though, as the scancodes can be compared just as well. As can be seen, the first keystroke always starts with a time of 0. The keystrokes after
that are measured relative to each other with the time differences in milliseconds. The first line of the file lists the total number of keystrokes, that is, total number of corresponding up- and down pairs. Therefore there will be 56 keystroke events in this particular file.

Most of the user data is nicely organized in the previously detailed method, but a significant portion of the database (around ten users) have smaller data sets than the common 16 or less imposter files available for them. This is probably because the users in question attended less sessions. An initial investigation of the data also revealed a great variability in the keystroke time values, as well as differing uses of non-printable characters such as left and right shifts.

<table>
<thead>
<tr>
<th>Character</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2aD</td>
<td>0</td>
</tr>
<tr>
<td>0x1eD</td>
<td>93</td>
</tr>
<tr>
<td>0x2aU</td>
<td>52</td>
</tr>
<tr>
<td>0x1eU</td>
<td>142</td>
</tr>
<tr>
<td>0x26D</td>
<td>3</td>
</tr>
<tr>
<td>0x12D</td>
<td>89</td>
</tr>
<tr>
<td>0x26U</td>
<td>7</td>
</tr>
<tr>
<td>0x12U</td>
<td>93</td>
</tr>
<tr>
<td>0x24D</td>
<td>44</td>
</tr>
<tr>
<td>0x24U</td>
<td>52</td>
</tr>
</tbody>
</table>

Table 5.1: Example keystroke file.

5.2 User-Dependent Classification

The user-dependent approach to keystroke classification maintains a unique model for every individual user. This is especially useful and powerful in the context of user verification. Since the user is given, the appropriate model can be immediately retrieved: a model which is optimized for this particular user. The research investigated user-dependent keystroking using a number of different variants:

- **Printable versus non-printable characters.** First of all, a distinction is made between printable and non-printable characters. Non-printable characters are those produced by pressing the left or right shift keys, the caps lock key or the backspace key among others. Since the database only guarantees that the printable characters of the name are the same for genuine and imposters, the extra information contained in non-printable characters will probably improve classification. A genuine user might always use the left shift key to produce capitals or special characters while an imposter might use the right shift key for example, a distinction that can be capitalized upon.

- **Latency, duration or total event data.** Furthermore, the classification methods were applied to latency data, duration data or the complete keystroke event data separately. Both the latency and duration data are smaller than the original keystroke event data and both can be derived from this data. This is done to investigate whether latency data is more useful for keystroke classification than duration data or not. Also, it will be interesting to see if these smaller data sets convey enough information for a classification with a similar error rate than classification based on the full event data.

- **Different classification methods.** Since keystroking involves relatively little preprocessing it can be primarily characterized as a classification problem. A number of different classification techniques were investigated with different error rates and time performances.

The following upcoming sections first discusses the preprocessing applied to obtain the latency, duration and event data sets and then moves on with the presentation of the different classification methods used. The imple-
ventions of all the methods, except the Simple Method, were provided by the Weka data mining platform\footnote{Freely available at \url{http://www.cs.waikato.ac.nz/ml/weka/}}. Some background information and theory on the method is provided along with the procedure followed to obtain the results. Since a unique model is trained for all the users, the discussion is on an individual user level without regard to the other users. After all methods are discussed, their results are presented in Subsection 5.2.4.

5.2.1 Preprocessing

The keystrokes will be used as the features to train the user model and to use that model for classification. In order to compare the keystrokes correctly they must be ordered: it is not sufficient to simply compare the entries in the feature vectors based on their index. Figure 5.2 illustrates three different keystroke vectors that produce the same sequence in three different ways. Comparing them based on their index, that is, line by line, would compare different keystroke events with each other and thus different features. This difference in ordering was widely observed in the database, even in keystroke data belonging to the same user.

Therefore a reordering of the keystroke events inside the vectors is required to make sure that every vector contains the same events in the exact same order. This reordering is achieved with the following steps:

1. **Determine a reference vector.** All the vectors are ordered against a reference vector which will contain the authoritative ordering. Any vector of the training set can be taken to be the reference vector.

2. **Convert the time differences to absolute times.** This means that the time value associated with a keystroke event no longer signifies its time difference to the previous event, but the time lapsed since the beginning of the keystroke sequence.

3. **Reorder the vector.** Give the keystroke vector the same ordering as the reference vector. If the vector under consideration contains an event not contained in the reference vector, add this event in the appropriate place in the reference vector based on its absolute time. On the other hand, if the reference vector contains an entry not present in the current vector, an ‘unknown’ value or ‘?’ is inserted at the appropriate place in the current vector.

4. **Compute the time differences based on absolute times.** Revert to the relative times between successive keystroke events by subtracting from all the time values the previous (absolute) time value.

The reordering process is illustrated in Figure 5.3. Example keystroke vectors V1 and V2 differ in their second and third keystroke events. These events are swapped to give V2 the same order as V1. Note that the time difference for the key ‘E’ down event becomes 100 since it occurred 100 after the key ‘K’ down event in the original V2 vector. The time difference for the key ‘K’ up event becomes negative as a result of the reordering. The reference vector V1 also contains a Shift event not present in V2, which is added to V2 with a time value of ‘?’.

Figure 5.2: Different keystroke vectors producing ‘KE’.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure5.2.png}
\caption{Different keystroke vectors producing ‘KE’.
}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure5.3.png}
\caption{Keystroke V2 reordered to match V1.
}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure5.4.png}
\caption{Keystroke V2 reordered to match V1.
}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure5.5.png}
\caption{Keystroke V2 reordered to match V1.
}
\end{figure}
The latency data can be easily computed from the keystroke event data by looking at successive *down* events. By making the time values of the events absolute, removing the *up* events and recomputing the relative times, the latency times are be obtained. Since a down event causes a character to be added to the entered string immediately and the database guarantees that all the strings are equal, the latency vectors do not need to be reordered. Care must be taken with different uses of non-printable characters such as left and right shift though. Duration data is obtained from matching *down* and *up* events belonging to the same keystroke, again using the conversion to absolute times. Like latency, no reordering should be necessary, although care must be taken with non-printable characters. Figure 5.4 illustrates the calculation.

**Figure 5.4:** Calculation of latency and duration.

### 5.2.2 Simple Approach

The first classification method is an original contribution of the author and is based on the mean and the standard deviation of the time values of a keystroke events and bears some similarity to a method proposed in [Hoc05]. The difference however is that this method uses a different scoring function and a different way to reach an *accept* or *reject* decision. It is trained with both genuine and imposter vectors.

#### Training

The method assumes that all the keystroke vectors, being either genuine or from imposters, are aligned according to the earlier discussed scheme and therefore contain the same events in the same order. Each of the vectors involved is of length $n$ and the method is trained with a selection of $m$ genuine keystroke vectors from the training set. The remaining other genuine and imposter vectors of the training set are used to determine the best threshold. The method then computes the average time $\mu_k$ and standard deviation $\sigma_k$ for each of the $n$ keystrokes using the time values from the $m$ vectors. Missing values due to different uses of non-printable keystrokes such as shifts are ignored when computing these means and standard deviations. The resulting means $\mu$ and standard deviations $\sigma$ are vectors of length $n$ and form the template for the user.

During verification, the test vector is reordered according to the stored template of the user. Keystrokes present in the test vector but not in the template are removed. The other way around, keystrokes present in the template but not in the test vector are given a very high value. After the vectors are both aligned, a distance is computed between the test vector $T$ and the template to produce a score. For each of the $n$ keystrokes a distance is computed using:

$$\text{score}_k = \exp \left( -\frac{(T_k - \mu_k)^2}{2\sigma_k^2} \right), 1 \leq k \leq n$$

(5.1)

where $T_k$ is the time value of the $k$th keystroke in the test vector. This obtained score is a number between 0 and 1. The more a keystroke from the test vector deviates from the mean, the lower its score will be. The keystrokes present in the template but not in the test vector are thus awarded with a score of zero. The $n$ obtained sums are then combined into a single score by a specific function. A number of functions were tested but it was found that the best results were obtained by taking the median of the scores as the overall score. If the score is more than some threshold, the user is accepted. This threshold is user-dependent and is computed during training with the other training vectors. The scores of those remaining vectors are computed and are used to determine the best
threshold for the user. In this case the goal was to minimize the overall error rate. This threshold, along with the mean and standard deviations forms the complete template of the user.

**Example**

![Training Set Example](image)

To illustrate the method a little example is presented here. The training set contains four genuine vectors (G1 to G4) and a single imposter vector I1, illustrated in Figure 5.5. The first three genuine vectors (G1 - G3) are used to compute the mean and average vectors \( \mu \) and \( \sigma \). In this example they turn out to be:

\[
\mu = \begin{bmatrix} 17.67 \\ 58.33 \end{bmatrix} \quad \text{and} \quad \sigma = \begin{bmatrix} 4.93 \\ 7.64 \end{bmatrix}
\]

Then the score vectors for each of the remaining vectors G4 and I1 in the training set can be computed:

\[
S_{G4} = \begin{bmatrix} 1.0 \\ 0.02 \end{bmatrix}, \quad S_{I1} = \begin{bmatrix} 0.3 \\ 0.22 \end{bmatrix}
\]

The overall scores of the vectors are then taken to be the medians. Since the vectors have an even number of entries, the median is the mean of the two middle values:

\[
s_{G4} = 0.51, \quad s_{I1} = 0.26
\]

In this particular example the threshold can be set to a value in the range \((0.26, 0.51]\) to achieve perfect classification on the training data. With real data such a nice separation is not very likely and there are bound to be imposter vectors with a higher score than genuine vectors. The threshold then has to be chosen to minimize the error.

**Method Application**

The keystroke database contains both genuine and imposter vectors for each user and the method was applied to each individual user. Three-quarters of both the genuine and imposter data were taken to be the training set, while the remaining quarters were used as a testing set to assess the method’s performance. Since the database holds 16 genuine vectors and 12 imposter files for a given user on average, the training and test sets were of size 21 and 7 respectively. The division into training and test was made randomly. During training 8 of the genuine users were used to compute the mean and average and the others were used to determine the threshold. After the means, standard deviations and thresholds were computed, each of the test vectors were classified. Cross-validation with four folds was used, which means that the experiment was run four times, each time using different three quarters of the data for training and one quarter for testing.

**5.2.3 Other Classification Techniques**

For the keystroking experiments a number of other classification techniques were used as well. These techniques, which are described next, are commonly used techniques for pattern classification in general and biometrics in particular. The implementations for these algorithms were provided by the Weka data mining platform in the form of standalone Java classes. In general, a classifier can be thought of as a system that takes a number of observations about the object that is to be classified. In the case of biometrics, these observations are the extracted features or attributes. The system then uses these observations to predict a class for the object. All classifiers need to be trained to some extent with data that is similar to the data they will see in their application environment.
5.2 USER-DEPENDENT CLASSIFICATION

Naive Bayes Classification

The Naive Bayes classifier is a probabilistic classifier and is suitable for very high dimensions, that is, a great number of features. The technique is relatively simple and often outperforms other more sophisticated methods. The technique is based on Bayes theorem:

\[ p(C_j|X_1,\ldots,X_n) = \frac{p(C_j)p(X_1,\ldots,X_n|C_j)}{p(X_1,\ldots,X_n)} \]

where \( C_j \) is a class and \( X_1,\ldots,X_n \) are the \( n \) observed features of the object. The conditional probability on the left is what the classifier is based on: given an observation \( X_1,\ldots,X_n \), the predicted class will be the one for which the conditional probability is the highest. The classifier computes this conditional probability by assuming that the conditional probabilities of the feature variables \( X \) are independent. That is, \( p(X_i|C_j) = p(X_i|C) \) for \( i \neq j \). Assuming independence, the formula on the right side of the equation can then be rewritten as:

\[ p(C_j|X_1,\ldots,X_n) = c p(C_j) \prod_{i=1}^{n} p(X_i|C_j) \]

where \( c \) is a constant to account for the denominator term \( p(X_1,\ldots,X_n) \). This term is not dependent of the class \( C \) and can be precomputed based on the features variables during training. In this form the model becomes much easier to compute since each \( p(X_i|C_j) \) can be computed separately. Notwithstanding the unreasonable independence assumption, Naive Bayes classifiers offer good performance at very little computation cost and requiring very little training. For the experimentation, the NaiveBayes Weka class was used.

Logistic Regression

Logistic regression is a variant of linear regression, but using a logistic function instead of a simple linear one. This type of regression is most commonly used as a binary classifier to predict class membership when there are only two classes (say genuine or imposter). Logistic does not assume a linear relation among the dependent (class) variable and the independent (feature) variables. The problem with linear regression is that it gives unrealistic predictions for very high or very small values of the independent variables; the line extrapolates outside the \([0, 1]\) range. In contrast, the logistic regression model approaches 0 and 1 at the low and high end respectively. The general model is:

\[ \log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 X_1 + \ldots + \beta_n X_n \]

where \( X_1,\ldots,X_n \) are the independent feature variables and \( \beta_0,\ldots,\beta_n \) are the regression coefficients. These coefficients are determined during the fitting of the training data to this model. During experimentation the Logistic Weka class was used for classification based on this technique.

Multilayer Perceptron

The multilayer perceptron is a feed-forward neural network that is trained using back-propagation training method. The MLP consists of an input and output layer and number of hidden layers consisting of perceptrons. During training, the observed features are applied at the input layer and the correct class is held at the output. The weights of the individual perceptrons are then updated to 'learn' a mapping from the features to the correct class. This update proceeds in a number of rounds or epochs and can be tuned by using different learning rates. The standard Weka MultiLayerPerceptron class was used and tested with different learning rates and number of epochs. Two variants of this algorithm were applied: the first variant performs a grid-search to determine the best number of epochs and learning rate to use for each user individually. This is variant 1. The seconds variant (number 2) takes a number of users (25) and calculates for each them the best number of epochs and learning rate. It then averages these rates and applies them to all other users when training an MLP for them. This does not require a very expensive grid-search for each user.
Support Vector Machines Classification

Support Vector Machines classification is based on a geometrical interpretation of the input. That is, the $n$ features of an observation are considered to be points in an $n$-dimensional space. Basic SVM is a binary classifier and assumes only two classes. The technique tries to create a $n-1$ dimensional hyperplane of that separates the data points. In the case of $n=2$ the hyperplane amounts to a line that divides the data points according to their class. SVM searches for the hyperplane that has the biggest margin, which means the distances between the points of either class and the hyperplane are maximized. The training data is used as the data points and it may be necessary to exclude some points in order to find a hyperplane that separates the data.

More advanced variants of SVM expand on the biggest margin concept by dealing with data which cannot be divided perfectly by a hyperplane. These soft margin methods still look for the biggest margin of a dividing hyperplane but also allow for values on the wrong side of the plane. These values add a penalty value to a particular hyperplane and the problem then becomes a search for the maximum margin and smallest penalty, between which a tradeoff exists. Typically the total penalty value is multiplied by a constant $C$. Even more advanced approaches use special kernel functions to map the feature space to another space. By ‘bending’ the feature space, it becomes possible to create a completely separating hyperplane. There are numerous kernel functions in use but for this research only the common radial basis function (RBF) kernel was used.

Weka offers support vector machines in the form of the SMO class which also features various kernels. As said, only the RBF kernel was used. A grid-search for the optimal parameters of the SVM was used to train the best classifier for each user individually, suggested by an approach taken in [Hsu06]. The RBF kernel has a single parameter $\gamma$, combined with $C$ penalty constant yields a two-dimensional search.

Tree-Based

Tree-based classification attempts to create a decision tree on the training data in order to classify future instances. Each internal node in the represents a decision to be taken on an observed feature and the leaf nodes represent a decision (class). During the training of such a tree, the features which have the most differentiating power are chosen (and placed at the top of the tree). The C4.5 algorithm is the most commonly used algorithm of this class and the Weka equivalent of it, called J48, was used during experimentation.

5.2.4 Results

After applying the methods in all the possible scenarios a number of results were obtained. First, the methods were applied to the keystroke data consisting of only the printable characters using either latency, duration or event based data. Then the methods were applied to all the keystroke data, including the non-printable characters. It is believed that the approach that includes non-printable characters as well offers better classification. Not only because it can use more information, but there exists considerable differences between users in the way they use these characters that can be capitalized upon. For instance, some users might use only the left shift key, while other users only the right one. Also the way a user makes capital letters might aid the classification. The results of both the printable- and all-characters approaches were obtained with custom scripts using the various Weka classifiers on a dual 3 GHz Intel Xeon machine with 4 GB or RAM. Four fold cross-validation was applied in all cases as well.

Printable Characters

The results for the data based on only printable characters are tabulated in Table 5.2. For each classification technique it displays the average and standard deviation of the error rate obtained for all the users, as well as the training time per user. As can be seen, the support vector machines classification offers the best performance, although it is also quite computationally intensive because of the required grid-search. Without such a search for the best parameters the technique gave very poor results. Furthermore it was discovered that there was no combination of values for the parameters that could be applied to all the users without seriously effecting the error rate.
From this data it also appears that duration-based data is less usable than latency-based data. The results are slightly worse than those obtained in a work by [Oba97] for the duration data. In that paper also the inter-key data, or time difference between the up event of the first key and down event of the second key, is considered. While in this research the latency, which is the time difference between successive down events, is used. In both researches, event-data, of which both the duration and latency data are subsets, offers the best basis for a classification. This is probably due to the fact that event-data contains more information. It is likely that the better results obtained in [Oba97] are due to the fact that there was much more training data available per user.

In all, the results are rather poor. Not only are most error rates in the double digits, the standard deviations of these error rates are also quite substantial. These error rates are not normally distributed since they lie fairly close to zero (less than one standard deviation). This suggests very high error rates for certain individual users.

### Table 5.2: Results for the user-dependent classification using only printable characters.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Method</th>
<th>Error Rate</th>
<th>SD of Error</th>
<th>Train Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>Support Vector Machines</td>
<td>10.87%</td>
<td>7.95</td>
<td>16.37 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 1</td>
<td>13.28%</td>
<td>8.61</td>
<td>20.01 sec.</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>15.22%</td>
<td>9.27</td>
<td>0.11 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 2</td>
<td>16.14%</td>
<td>10.63</td>
<td>2.98 sec.</td>
</tr>
<tr>
<td></td>
<td>J48 Tree</td>
<td>21.09%</td>
<td>9.67</td>
<td>0.06 sec.</td>
</tr>
<tr>
<td></td>
<td>Simple Method (median)</td>
<td>21.56%</td>
<td>13.63</td>
<td>1.43 sec.</td>
</tr>
<tr>
<td></td>
<td>Logistic Regression</td>
<td>22.58%</td>
<td>13.63</td>
<td>0.21 sec.</td>
</tr>
<tr>
<td>Latency</td>
<td>Support Vector Machines</td>
<td>7.86%</td>
<td>6.1</td>
<td>16.36 sec.</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>8.69%</td>
<td>6.89</td>
<td>0.11 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 1</td>
<td>9.04%</td>
<td>6.56</td>
<td>22.19 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 2</td>
<td>11.45%</td>
<td>8.24</td>
<td>2.96 sec.</td>
</tr>
<tr>
<td></td>
<td>Simple Method (median)</td>
<td>13.23%</td>
<td>10.58</td>
<td>1.69 sec.</td>
</tr>
<tr>
<td></td>
<td>J48 Tree</td>
<td>15.85%</td>
<td>9.48</td>
<td>0.06 sec.</td>
</tr>
<tr>
<td></td>
<td>Logistic Regression</td>
<td>16.06%</td>
<td>10.21</td>
<td>0.2 sec.</td>
</tr>
<tr>
<td>Event</td>
<td>Support Vector Machines</td>
<td>5.97%</td>
<td>5.85</td>
<td>16.55 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 1</td>
<td>7.24%</td>
<td>6.04</td>
<td>27.17 sec.</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>7.35%</td>
<td>5.59</td>
<td>0.18 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 2</td>
<td>8.55%</td>
<td>6.86</td>
<td>2.94 sec.</td>
</tr>
<tr>
<td></td>
<td>Simple Method (median)</td>
<td>12.24%</td>
<td>10.95</td>
<td>2.99 sec.</td>
</tr>
<tr>
<td></td>
<td>J48 Tree</td>
<td>12.63%</td>
<td>9.78</td>
<td>0.23 sec.</td>
</tr>
<tr>
<td></td>
<td>Logistic Regression</td>
<td>16.78%</td>
<td>9.38</td>
<td>0.06 sec.</td>
</tr>
</tbody>
</table>

### Printable and Non-printable Characters

The results for the keystroking based on all the characters are displayed in Table 5.3 in the same way as they were for the printable characters. The results are broadly similar, although the error rates are lower than with the printable characters-only approach. This is probably due to the fact that the classifier is based on more attributes (and thus more information) and the discriminatory qualities of the non-printable characters as hypothesized earlier. Again the best result are obtained with the support vector machines even getting to an error rate under five percent. In fact, of all the approaches tried in this chapter, this was the best result. As can be seen, the added computation cost of using more features clearly outweighed by the benefits of the higher classification performance.
5.3 User-Independent Classification

The unimodal approach tries to classify all the users using a single model. This implies that certain common features have to be extracted from individual keystroke sets. One of those approaches is based on digraphs, which are sequences of two successive keystrokes such as ‘E-N’. The digraph approach will be investigated in Subsection 5.3.1. Another single-model approach is based on typing rhythms. The different keystrokes can be categorized into different ‘time rhythms’ such as 50-100ms, 100-200ms etc. Subsection 5.3.2 will go into more detail about this. A final method investigated is the time-ranking approach. Basically the keystrokes are ordered by their time values and the average rank of a keystroke in that ordering is used in calculating the distance between different vectors. Subsection 5.3.3 will delve deeper into this method.

5.3.1 Digraph Approach

Keystroking biometrics based on digraphs is one of the earlier and extensively studied methods [Leg88] [Mon97] [Mon99]. Most techniques gather training data from users when they type a large piece of text specifically crafted to yield a representative set of digraphs. While the user is typing the text, the digraphs are captured along with their time characteristics. The most commonly used time value is the down-to-down time, that is, the time between the first and the second down events also referred to as latency. Other metrics used are the total duration of the whole digraph, duration of the first keystroke of the digraph and the inter-key latency, that is the time between the first up event and the second down event. Since latency is most commonly used, it will be adopted as the timing metric of digraphs in this research as well.

Considering the alphanumeric characters, whitespace characters, shifts and other special characters, the num-

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Method</th>
<th>Error Rate</th>
<th>SD of Error</th>
<th>Train Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration</td>
<td>Support Vector Machines</td>
<td>7.03%</td>
<td>6.68</td>
<td>15.77 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 1</td>
<td>10.14%</td>
<td>8.02</td>
<td>21.82 sec.</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>11.35%</td>
<td>7.79</td>
<td>0.14 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 2</td>
<td>12.56%</td>
<td>9.57</td>
<td>3.16 sec.</td>
</tr>
<tr>
<td></td>
<td>Logistic Regression</td>
<td>17.97%</td>
<td>12.79</td>
<td>0.23 sec.</td>
</tr>
<tr>
<td></td>
<td>Simple Method (median)</td>
<td>18.77%</td>
<td>12.73</td>
<td>1.45 sec.</td>
</tr>
<tr>
<td></td>
<td>J48 Tree</td>
<td>20.58%</td>
<td>9.13</td>
<td>0.06 sec.</td>
</tr>
<tr>
<td>Latency</td>
<td>Support Vector Machines</td>
<td>6.37%</td>
<td>5.04</td>
<td>15.8 sec.</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>8.09%</td>
<td>6.05</td>
<td>0.15 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 1</td>
<td>8.81%</td>
<td>6.05</td>
<td>24 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 2</td>
<td>10.45%</td>
<td>7.35</td>
<td>3.13 sec.</td>
</tr>
<tr>
<td></td>
<td>Simple Method (median)</td>
<td>12.35%</td>
<td>12.46</td>
<td>1.19 sec.</td>
</tr>
<tr>
<td></td>
<td>Logistic Regression</td>
<td>12.9%</td>
<td>7.45</td>
<td>0.22 sec.</td>
</tr>
<tr>
<td></td>
<td>J48 Tree</td>
<td>15.27%</td>
<td>8.16</td>
<td>0.06 sec.</td>
</tr>
<tr>
<td>Event</td>
<td>Support Vector Machines</td>
<td>4.91%</td>
<td>4.72</td>
<td>16.19 sec.</td>
</tr>
<tr>
<td></td>
<td>Naive Bayes</td>
<td>7.11%</td>
<td>6.26</td>
<td>0.23 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 1</td>
<td>7.14%</td>
<td>5.23</td>
<td>30 sec.</td>
</tr>
<tr>
<td></td>
<td>Multilayer Perceptron 2</td>
<td>9.6%</td>
<td>6.86</td>
<td>2.79 sec.</td>
</tr>
<tr>
<td></td>
<td>Logistic Regression</td>
<td>11.38%</td>
<td>7.84</td>
<td>0.25 sec.</td>
</tr>
<tr>
<td></td>
<td>Simple Method (median)</td>
<td>12.51%</td>
<td>10.21</td>
<td>2.09 sec.</td>
</tr>
<tr>
<td></td>
<td>J48 Tree</td>
<td>14.86%</td>
<td>8.82</td>
<td>0.07 sec.</td>
</tr>
</tbody>
</table>

Table 5.3: Results for the user-dependent classification using all characters.
ber of possible digraphs is rather great. Since a digraph-based classification system considers the individual digraphs as attributes for its classification, a great number of digraphs has a detrimental effect on the duration of the training and classification. Furthermore, there is a great asymmetry in the frequencies of the digraphs. For instance the digraph 'QZ' is unlikely to yield any instances, while 'TH' will have a high frequency, at least for the English language. Therefore most systems use only the most frequent digraphs. That is possible because most approaches tend to be based on training data of hundreds or thousands of words, where it is likely that a common subset of all the digraphs exists which is represented in all the training data of the users.

In this research however the training sets are rather small (on average 27 keystrokes). Therefore the number of digraphs considered will be varied ranging from all the digraphs encountered in the keystroke database down to only 10. Reducing the number of digraphs used will definitely increase the time performance of the training and classification but there is a risk of 'shutting out' users. Because the database consists only of the typed name of the user, there is a risk that he or she does not feature the most frequent digraphs, or only one or two. In other words, it is rather unlikely that there is set of digraphs shared by all users in this particular database. In fact, the most common digraph present in the database was only featured by 72 of the 132 users, with the majority of the digraphs having a much lower representation.

Besides different digraph/attribute sizes, different classifiers will be applied to the data, again using Weka. Since all the users are known in the data set it is much larger than than in the individual, user-dependent approach, and classification is expected to take much more time. Because of the bigger data set, ten-fold cross-validation will be applied instead of the four-fold cross-validation applied earlier. With preprocessing scripts a number of data sets were created, each with a different number of digraphs.

**Preprocessing**

The preprocessing of the keystroke data into digraphs was done in the following steps:

- **Determine total digraphs set.** The keystroke data of all the users is scanned for digraphs in order to come up with the total set of digraphs present in the database. The number of users that exhibit a particular digraph is taken as that digraph’s frequency.

- **Determine user digraphs.** On a per-user basis, all timing data is gathered for the different digraphs. Since all user typed their a number of times this results in multiple time values available per digraph.

- **Create overall digraph data.** All the individual user digraph data is gathered into a single CSV file for easy processing. The digraphs and class (username) serve as columns while the different instances of time values serve as rows. If a user does not feature a particular digraph, an ‘unknown’ value is inserted. The digraph columns are sorted according to their frequency (higher frequency first).

By removing columns from the file obtained, different data sets are created with a different number of digraphs. The CSV files can be readily used by Weka for classification and analysis. These data files were very sparse and especially when the number of digraphs was low, some users were only represented with one or two values. A minimum of 15 different digraph attributes was found to be a minimum for which all users were still represented. Files with lower numbers were inevitable not representing a large number of users.

**Experimentation and Results**

The results of the experiments with digraph-based data are displayed in Table 5.4. Only the Bayes classification was used since the other techniques gave either much worse results or took way too long to finish their training. By ‘commonality’, the minimum number of users which feature a particular digraph is meant. If this value is zero, then all the possible digraphs are included and if it is 32 then only the digraphs which are present in the data of at least 32 users is included. The higher this number, the lower the number of attributes and the lower the number of represented users as can be seen in the train- and test size columns. Beyond 32, the number of users that was not represented started to grow quickly.

---

2Comma separated values
The error rates that are achieved by this technique are rather high as compared to the results obtained with the user-dependent techniques. Surprisingly, the error rate is more or less independent of the number of attributes and this is probably due to the sparsity of the data on which the Bayes classifier manages to capitalize. Not only have different users different values for their attributes, they also only have values for differing subsets of those attributes. Another problem of this approach is that it is not scalable with small training data sets: if the number of users grows, more and more digraphs (attributes) are required to represent them all. In the end this might require all the possible combinations of digraphs which is probably prohibitively expensive to train on. Also note that a retraining is required after the addition of a new user.

<table>
<thead>
<tr>
<th>Commonality</th>
<th>Error Rate</th>
<th>FAR</th>
<th>Trainsize</th>
<th>Testsize</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19.85%</td>
<td>28.9%</td>
<td>2878</td>
<td>932</td>
<td>379</td>
</tr>
<tr>
<td>4</td>
<td>20.06%</td>
<td>28.9%</td>
<td>2878</td>
<td>932</td>
<td>263</td>
</tr>
<tr>
<td>8</td>
<td>19.96%</td>
<td>28.9%</td>
<td>2878</td>
<td>932</td>
<td>199</td>
</tr>
<tr>
<td>12</td>
<td>19.44%</td>
<td>27.7%</td>
<td>2872</td>
<td>931</td>
<td>161</td>
</tr>
<tr>
<td>16</td>
<td>19.87%</td>
<td>28.0%</td>
<td>2872</td>
<td>931</td>
<td>135</td>
</tr>
<tr>
<td>20</td>
<td>19.76%</td>
<td>27.3%</td>
<td>2872</td>
<td>931</td>
<td>117</td>
</tr>
<tr>
<td>24</td>
<td>18.90%</td>
<td>25.5%</td>
<td>2864</td>
<td>931</td>
<td>95</td>
</tr>
<tr>
<td>28</td>
<td>19.35%</td>
<td>26.2%</td>
<td>2862</td>
<td>930</td>
<td>81</td>
</tr>
<tr>
<td>32</td>
<td>18.68%</td>
<td>25.6%</td>
<td>2840</td>
<td>921</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 5.4: Results of the digraph-based approach using a Naive Bayes classifier.

5.3.2 Rhythm-based Approach

The rhythm-based approach is based on a method presented in [Hoc05] and works by assigning digraphs into rhythm-classes. Just as with music, the rhythm of a digraph is related to its duration or time value. In the same way that music notes can be classified as being 'very short', 'short' or 'long', the digraphs are also grouped into several rhythm classes:

- $t < 30$: class 1
- $30 < t < 30$: class 2
- $70 < t < 100$: class 3
- $100 < t < 200$: class 4
- $200 < t$: class 5

For each user, the rhythm classes of the digraphs of the training vectors are computed. The average class of each digraph then constitutes the users template vector, which will be compared against vectors from a test set. When matching, the distances of matching digraphs of the template and the test vector are summed to yield an overall score. This distance is defined as being the difference in rhythm class: if a digraph has class 2 in the template but class 5 in the test vector, then the distance will be 3. If the training (template) vector features a digraph but the test vector does not, a penalty value is added to the sum instead. The score is then compared with a threshold and if it surpasses that, the test vector will be rejected and accepted otherwise. In the original paper only the down-to-down digraph times were used, but in this research also the individual keystroke values were tested.
Experimentation and Results

The rhythm-based method can be tweaked through two of its parameters which are threshold used and the penalty assigned for missing values. The best value for these parameters was obtained by performing a grid-search on them, that is, testing all the possible combinations of parameter values from certain ranges. Both for the threshold as the missing value penalty integer numbers were drawn from the $[0, 50]$ range. The results of the grid-searches and the subsequently lowest error rates are depicted in Figures 5.6 and 5.7 for keystroke and digraph data respectively.

![Figure 5.6: Grid-search for lowest error rate using keystroke data.](image)

![Figure 5.7: Grid-search for lowest error rate using digraph data.](image)

As can be clearly seen from the two surface plots, the average error rate is largely dependent on the threshold used and not on the missing value penalty. Since a vector, not belonging to the user against whose template it is being matched, will tend to have a large number of missing values (because it has different digraphs or keystrokes), its computed distance will quickly surpass the threshold, no matter how low the missing value penalty. Except of course when the missing value penalty is very close to zero. But a missing value penalty of zero will also lead to many false positives and the FAR will then dominate the overall error rate.

The best observed error rate of 13.33 percent for the keystroke data is still rather high as compared to the user-dependent techniques mentioned earlier in this chapter. On the other hand, the technique requires very little computation and is also linear with respect to the number of users (under the assumption that there is a maximum number of keystrokes, as is the case with password data for instance). This in contrast with the digraph-based technique. Perhaps the technique can be further improved by selecting more and different rhythm classes.

The corresponding ROC curves of the rhythm-based approaches based on both digraph and keystroking data are plotted in Figure 5.8. As can be seen, the curve belonging to the keystroke data has more area underneath it and is therefore better. This is probably due to the fact that by using single keystrokes there is twice as much information to base a classification on.

5.3.3 Time-Ranking Approach

The time-ranking approach to keystroking is based on [Hoc05] and [Ber02]. The method works by ordering the entries in the vectors by their time value and storing the average ranks of the entries as the template. During verification or identification, a comparison of the ranks is made between the template vector and a given test vector using a rank correlation method. The rank correlation method expresses the agreement between the ranks in terms of a single number which serves as the score. A decision can then be reached by comparing this score against a threshold, or, in the case of identification, picking the template which had the highest score against the test vector.

Preprocessing

The preprocessing applied to the raw keystroke data was broadly similar to the procedure used in Subsection 5.3.2. Each keystroke file is transformed into vector of tuples. Each tuple contains the keystroke or digraph’s name and the associated time value. For digraphs, the down-to-down time was taken while for the keystrokes the delta time
with respect to the previous keystroke (latency) was used. This vector of tuples was then ordered (ranked) by the time values in ascending order. The training vectors are combined into a single vector which will serve as the user’s template. Each of the entries of this template vector are computed by averaging the time values for each keystroke or digraph. A quarter of genuine files and all the imposter files were assigned to the overall test set, while the remaining genuine files were committed as training data. Figure 5.9 illustrates the basic process.

Figure 5.8: ROC curves for the rhythm-based approach using either keystroke- or digraph-based data. The curve is created by varying the threshold while using the missing value penalty for which the lowest error rate is achieved.

Figure 5.9: Time-ranking process. Vectors T1, T2 and T3 form the training set and are used to compute a template vector which holds the average ranks for the entries A, B, C and D. This template is then used in a rank comparison against a test vector yielding a score.

Rank Correlation Methods

There were three different methods used to calculate the rank correlation between the training data of a user and a given test vector. In the case of time-ranking, the rank correlation is considered to be the score when matching a training and test vector. The first two methods were taken from [Hoc05] while the third method (Kendall tau coefficient) comes from the field of statistics. The three methods were extended slightly to accommodate differences in keystrokes/digraphs between the rankings under consideration. These amendments are discussed per method. If these provisions would not have been taken, then a comparison rankings would have to be based on the common subset of keystrokes/digraphs, which might be empty.
**Euclidian Distance.** The first distance measurement used to test the correspondence between two rankings is the Euclidean distance between them. The distance $D$ between two rankings $a$ and $b$ with $n$ items is calculated as:

$$D = \sqrt{(r_{a1} - r_{b1})^2 + (r_{a2} - r_{b2})^2 + \ldots + (r_{an} - r_{bn})^2}$$ (5.2)

where $r_{ai}$ is the ranking of the $i$th item in ranking $a$. A distance of 0 indicates a perfect agreement of the ranks, while a higher value indicates a poor correlation.

In order to cope with rankings of different length or different items, the basic Euclidian distance calculation was extended. The training vector is compared, keystroke by keystroke, against a test vector. If the keystrokes match (they have the same name) then the difference is simply calculated. In the event that the training vector contains a keystroke not present in the test vector, an alternative 'missing value' is taken as the distance instead. This value can be considered as a penalty and its optimal value was determined during experimentation.

**Spearman Rank Correlation Coefficient.** The Spearman Rank Correlation Coefficient, denoted with $\rho$ expresses the correlation in rank between two rankings in the form of a single value. The coefficient is calculated as follows:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$ (5.3)

where $d_i^2$ is the difference in ranking of an item $i$ in the two different rankings and $n$ is the number of items. If item $i$ is ranked similarly in the two rankings, then $d_i^2$ will be lower and $\rho$ will subsequently be higher. If all the items have the same ranking then the coefficient will be 1, indicating a total rank correlation. The closer the coefficient is to 0, the more disagreement is present between the two rankings. To cope with different ranking vectors, the number of keystrokes present in the training ranking but not in the test ranking is counted. If this count exceeds a certain limit, the correlation is considered to be 0.

**Kendall Tau Coefficient.** This coefficient expresses the correlation in rank between two rankings or in this case ranked keystroke or digraph sequences in terms of a single number. The coefficient is traditionally designated by the Greek tau letter and is calculated as follows:

$$\tau = \frac{4P}{n(n-1)} - 1$$ (5.4)

where $n$ is the number of ranked items. $P$ is the sum, over all the items, of items ranked after the given item by both rankings. The calculation of $P$ and $\tau$ is best illustrated by means of a small example. Consider a group of four individuals (A, B, C, D) which are ranked in two different categories, age and weight. For the age category the ranking is as follows: A = 1, B = 2, C = 3 and D = 4. For the weight category the ranking is different: A = 3, B = 4, C = 1 and D = 2. Calculating $P$ now proceeds per user, starting with person A. For A, there are three other persons (items) which rank behind him in age and there is only one person who ranks behind him in weight, so A’s contribution to the $P$ sum is 1. For the other persons the contributions are 0, 1 and 0 respectively. $P$ is then $P = 1 + 0 + 1 + 0 = 2$. The Kendall tau coefficient can then be calculated as follows:

$$\tau = \frac{4P}{n(n-1)} - 1 = \frac{8}{4(4-1)} - 1 = 0.25$$

The $\tau$ coefficient takes on a value in the range $[-1, 1]$ where 1 indicates a total rank correlation, that is, a total agreement of the ranks between the two rankings. A value of -1 expresses a total disagreement in the rankings, effectively meaning that the first ranking is the reverse of the second. If the rankings are not correlated the coefficient will be 0 on average. Differences in the items themselves (not their ranking) are dealt with in the same way as with the Spearman Rank Coefficient: by limiting the number of missing items allowed.
Experimentation and Results

All three methods have two parameters that can be varied. The first being the maximum number of missing values allowed in the test vector (or in the case of the Euclidean distance, a penalty value) and the threshold to use. Note that in order to be accepted, the score of a test vector must be greater than this threshold. In order to determine the best values for these two parameters, a grid-search was performed in much the same way as done with some of the user-dependent techniques. The time-ranking method was also applied to both single keystroke data (latency) and digraph data (down-to-down).

<table>
<thead>
<tr>
<th>Distance Method</th>
<th>Single Keystroke</th>
<th>Digraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Euclidean Distance</td>
<td>18.81 %</td>
<td>15.93 %</td>
</tr>
<tr>
<td>Spearman Coefficient</td>
<td>21.83 %</td>
<td>8.75 %</td>
</tr>
<tr>
<td>Kendall Tau</td>
<td>25.19 %</td>
<td>12.00 %</td>
</tr>
</tbody>
</table>

Table 5.5: Error rates for the various time-ranking methods, applied on different types of keystroke data.

The results using the Euclidian distance for both the keystroke- and digraph-based data are listed in Table 5.3.3. From the various grid searches conducted to determine the best missing value penalty or threshold it became clear that the latter had a much more profound impact on the error rate achieved than the former. In general, setting the threshold closer to zero led to lower error rates. Also, the recognition based on digraph data was better in all cases than by using single keystroke data. As can be seen from the ROC curves in Figure 5.10, the best results can be obtained by using digraph-based data in combination with the Spearman correlation coefficient.

![ROC curves for the time-ranking methods](image)

Figure 5.10: ROC curves for the different time-ranking methods using either keystroke- or digraph-based data. The curve is created by varying the threshold while using the missing value penalty for which the lowest error rate is achieved.

The results of the ranking-based approach are rather low compared to other techniques explored earlier in this chapter. This is probably due to the fact that not all information in the keystroke vectors is used; only the ranks are considered and the actual time values are used only indirectly. The assumption that the ranking of the keystroke times is the most discriminatory aspect of the keystroke data is probably to crude. It misses a lot potentially useful extra information contained in the keystrokes. Since the data set is already very limited, reducing it even further to only rank information might limits the biometric performance.

Furthermore, an outlying value for a particular digraph will offset a significant number of other digraphs. In the
worst-case, if a digraph, which is ranked first in the template, has a very large outlying value in the test vector, then this digraph will be ranked last in the test vector with all other digraphs shifting one rank lower. Interesting to see is that digraph-based ranking performs much better than keystroke-based ranking. Creating the template vectors and verifying them against test vectors however takes little time, especially as compared to the full digraph-based approach described in [5.3.1] or some of the user-dependent approaches. The computation time is also linear in the number of users in the database (assuming the number of digraphs or keystrokes is bounded, as is the case with passwords keystroke data).

5.4 Keystroke Module

In order to use the Bio-SPHERE platform for multimodal biometrics, a keystroking module will be created for it. This module will be called bsmod_keystroke.so and will incorporate both the pre-processing (reordering) techniques presented earlier, as well as the several classification techniques. A choice is made to support only user-dependent classification and to forgo user-independent keystroking. The primary reason for this is the inherently unscalable characteristics of maintaining a single model for all users. Retraining the model every time a new user is enrolled or updated will take more and more time. Also, user-independent keystroking tends only to perform better when more attributes are used (for example by using more digraphs) which exacerbates the scalability problem. User-independent keystroking also gives very low performance for a significant part of the population, effectively meaning that those users will not be able to use the system.

It makes sense to again make use of the Weka platform and its set of Java-based classifiers in creating the module, rather than coming up with proprietary classification implementations. During the keystroke experimentation some helper programs were created based on Java and Weka which can be readily wrapped into a module. Therefore, the keystroking module will consist of two distinct parts: the module itself, interfacing with the Bio-SPHERE server and handling the requests it is passed, and a Java program that implements the preprocessing and classification based on Weka. Figure 5.11 illustrates the structure of the module.

![Figure 5.11: Structure of the keystroke module.](image)

On the left, the Bio-SPHERE server, which has loaded the keystroking module, passes down service requests designated for the keystroking module. The module interprets the requests and sends appropriate commands through a pipe to the Java companion program using a simple line-based protocol. The companion program in turn forms the core of the keystroking: it enrolls and removes users, adds training data and classifiers to them and verifies given test vectors. Since a user-dependent approach is taken, every user has its own set of training vectors and personally trained classifiers. Persistent storage is achieved by saving the user’s object instances as a file using Java’s serialization facilities. After the companion program has worked its magic, it creates a temporary file with status information which is read in turn by the module. The module then finally relays this status information to the Bio-SPHERE server in the form of a service response message.

As part of the keystroking module’s initialization routine it starts up the companion program and sets up a write-only pipe to it. When the module is unloaded, it tears down this pipe and with it the companion program on the other end. By using this pipe, the keystroking module can send data to the companion program as if it were a file using calls like fprintf. The companion program receives this data on its standard input. During the
handling of service requests to it, the module forwards the (transformed) requests to it’s companion using a simple line-based protocol which consists of the following six commands:

- **ENROLL <username>**
  A new user will be enrolled. This means that a user object will be created, still devoid of any training vectors and classifiers. It will be serialized and stored in a file called username.dat. Note that each command is ended with a newline.

- **REMOVE <username>**
  The named user will be removed from the database, which is done by deleting his or her data file.

- **ADD <username> <class> <vectordata>**
  Add a training vector to the named user. The vector has a certain class: it can be either genuine or imposter data. The vector will be reordered by the companion program and then added to the known training vectors of that user. It is an error if the named user does not exist. The vector data comes in the form of alternating whitespace (not newline) separated, keystroke and time values, such as: 0x12D 10 0x1eD 20 ...

- **TRAIN <username> <method>**
  Train a classifier for the named user using the training vectors gathered for him or her so far. The companion program incorporates several classification techniques which must be specified. Choices are bayes, j48, logistic, mlp or svm. Note that several different classifiers can be trained for the same user.

- **VERIFY <username> <method> <vectordata>**
  Verify if the vector corresponds to the named user. In other words verify the given user, using the named classification technique. The vector data is in the same format as for the ADD command. The program will produce an accept or reject decision.

- **VERIFYTHRESHOLD <username> <method> <vectordata>**
  This command is very similar to the VERIFY command but instead of a decision, it will only produce a score in the range [0, 1].

The Java companion program itself will be relatively straightforward and its class diagram is illustrated in Figure 5.12. It is contained in .jar file called keystroke.jar. The KeystrokeCompanion class contains the Main method which contains a continuous loop. In this loop, commands are read from standard input, executed and the results are put into temporary files. The KeystrokeCompanion class loads, manipulates and stores KeystrokeUser objects depending on the commands received. The KeystrokeUser objects themselves represent a single user and contain his or her training data and classifiers trained with that data. The KeystrokeUser object also maintains a reference vector to maintain the same ordering of keystrokes among all training vectors.

The KeystrokeVector class represents a single keystroke vector, consisting of at least one KeystrokeTuple. The KeystrokeVector class also contains some utility methods to assist in the ordering of the data. The KeystrokeTuple class is a simple container class for the keystroke and its associated time value. This class can also represent keystrokes which have an unknown value. The ClassifierInstance class represents a single Weka Classifier class and provides a wrapper around the Weka classifiers. It sets the appropriate options and filters when building the classifier. It includes several methods of the type makeXClassifier(), where X is a particular classification technique.
Figure 5.12: Class diagram of the keystroke companion program.
Chapter 6
Multimodal Experiments

This chapter will describe the various multimodal experiments undertaken as part of the Bio-SPHERE project to investigate if and how much multimodal biometrics improves the error rates. The experiments amounted both to match level as well as decision level fusion using two different biometric modes: keystroking and hand geometry. Keystroking was treated in Chapter 5 while hand geometry is a biometric based on the various dimensions of a person’s hand and fingers. It will be further described in Section 6.1. For the experimentation the previously described keystroking module will be used as well as a new module for the hand geometry biometric. The fusion module described in Section 4.5 is used for the actual fusion.

The experimentation will encompass both match level fusion and decision level fusion using all the methods described in Sections 4.3 and 4.4. Since the number of distinct biometric modes will be limited to just two, there will also be some intramodal fusion experimentation. Intramodal fusion uses the same biometric modes, such as keystroking, but uses two or more different classifiers in order to lower the error rates. After the biometric modes have been presented, decision level fusion will be presented since it is simpler than match level fusion, which comes after it.

6.1 Biometric Modes

The keystroking module was trained with all the users from the database, but using only a limited number of techniques. Since the lowest error rates were achieved with full keystroke event data, that data was used in favor of latency-based or duration-based data. Three quarters of this data were entered into Bio-SPHERE for training purposes while the remaining quarter was used later on for testing. Before using the module in multimodal experiments, measurements were taken to determine the FAR, FRR and error rates. Table 6.1 lists the trained classifiers and their corresponding error rates. The support vector classifier used again made use of a grid-search to find optimal parameters but less elaborate than in the previous chapter. This explains the higher error rates. Only three different classifiers were trained to limit the number of possible combinations possible for multimodal biometrics. Trying every combination of classification and fusion method is unnecessary and too time-consuming in order to get general results. The table also displays the average training time per user and the average time taken by a single verification. The addition of training vectors takes little time: on average about 200 microseconds. The results were obtained by testing with 518 genuine vectors and 368 imposter ones.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>FAR</th>
<th>FRR</th>
<th>Error Rate</th>
<th>Train time</th>
<th>Verify time</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48</td>
<td>20.9 %</td>
<td>10.4 %</td>
<td>14.8 %</td>
<td>220 msec.</td>
<td>256 μsec.</td>
</tr>
<tr>
<td>Logistic</td>
<td>20.1 %</td>
<td>4.9 %</td>
<td>11.2 %</td>
<td>430 msec.</td>
<td>310 μsec.</td>
</tr>
<tr>
<td>SVM</td>
<td>17.9 %</td>
<td>3.8 %</td>
<td>9.7 %</td>
<td>31 sec.</td>
<td>415 μsec.</td>
</tr>
</tbody>
</table>

Table 6.1: Error rates and computation times for different keystroking classifiers.

For the multimodal experiments use will be made of not only the keystroking biometric but also of other bio-
metrics. The same database used for the keystroking also features several other modes, but due to circumstances the only other mode that was used for multimodal experimentation is an approach based on hand geometry. The other modes were not sufficiently developed enough for experimentation. This biometric is still in active development, but some preliminary results obtained were already good enough for multimodal experimentation.

Hand geometry is based on the various dimensions present in a human hand. These are the widths and lengths of the fingers and palm, but not the fingerprint or line pattern of the hand. The various dimensions that are measured by the system are depicted in Figure 6.1.

As stated before, the hand geometry biometric is still the subject of active research. As such it also not a single coherent system, but rather a collection of separated programs for pre-processing and feature extraction with classification fully done in Weka. Integrating these programs into a single module is therefore not suitable and the choice was made to use intermediate data with the extracted features. This intermediate data consisted of nicely formatted, comma-separated-value files with a fixed number of attributes, albeit for only a subset of the database (77 users). A module was wrapped around a collection of Weka classifiers using this data, in very much the same way as the keystroking module. The big difference between the two however is, that the keystroking module creates a model (and thus a data file) for each individual user, whereas a user-independent approach was taken for the hand geometry module.

The reason for training a user-independent classifier stems from the fact that the data only represents genuine users and no imposters. Using this data without precautions would lead to a classifier that only has a false reject rate as there are no imposters to reject. Therefore the hand geometry database was split in two: one half being the genuine users and the other half were all relabeled as belonging to the 'unknown' class. So if one of these unknown users is classified as one of the genuine users, we know we have a false positive. The hand geometry classifier is thus not a binary classifier, but a multi-class one. The newly obtained data set was then divided into three-quarters of training data and one quarter of test data, the training vectors were entered into Bio-SPHERE. The data consisted of a fixed number of attribute values, corresponding to hand features, and a class attribute. The resulting FAR, FRR and overall error rates obtained after training are displayed in Table 6.1. This time the training time is the time taken for the training of the complete training set. The testing was done using 128 genuine vectors and 150 imposter (class ‘unknown’) vectors.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>FAR</th>
<th>FRR</th>
<th>Error Rate</th>
<th>Train time</th>
<th>Verify time</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48</td>
<td>13.61 %</td>
<td>34.1 %</td>
<td>23.1 %</td>
<td>2.30 sec.</td>
<td>300 µsec.</td>
</tr>
<tr>
<td>Bayes</td>
<td>10.0 %</td>
<td>9.5 %</td>
<td>9.8 %</td>
<td>1.56 sec.</td>
<td>272 µsec.</td>
</tr>
<tr>
<td>Logistic</td>
<td>5.4 %</td>
<td>5.7 %</td>
<td>5.5 %</td>
<td>670 sec.</td>
<td>262 µsec.</td>
</tr>
</tbody>
</table>

Table 6.2: Error rates and computation times for different classifiers on hand geometry data.
6.2 Decision Level Fusion Experiments

There are many different variants when doing decision level fusion. First of all, the number and type of biometrics can be varied. Second, different classification methods can be employed per mode and of course there are also different fusion methods which can be used, some of which can be tuned by setting certain parameters. To keep the experimentation manageable, a choice was made to limit the number of different experiments. Most importantly, the fusions will only involve one or two different biometrics. As seen earlier, each of the two biometrics can use three different classification methods and four different fusion methods as described in Section 4.4. The experimentation was automated using various python-based scripts which made use of Bio-SPHERE’s python library [B.3].

When the keystroking and hand geometry biometrics are fused, or intermodal fusion, only the users represented by both biometrics can be used. The applied vectors have to be either both genuine or both from imposters. It is not very probable that a user who is appropriately enrolled in one biometric tries to spoof the other one at the same time. Of course, both vectors have to come from the same user. This is easy in the genuine case, but the imposter case is a bit different. For the keystroking biometric an imposter for a given user can be found directly, but the hand geometry data is less personal: it has no names attached to unknown (imposter) users. The decision was taken to pair up an ‘unknown’ user from the hand geometry with a random imposter from the keystroking.

6.2.1 Fusion With Two Biometrics

Using just two biometric modes has some complications for the fusion methods that can be used. First, the majority voting method makes no sense to use with just two modes: a required majority of 1 is equivalent to the Boolean-Or method and a majority of 2 is equivalent to the Boolean-And method. The decision weighting method also does not make sense with just two biometrics. Since the weights depend on the EER (or some other reliability indicator) one biometric will be assigned a higher weight than the other (since it is very unlikely that they have the same EER). Now there are two cases. If the threshold used is lower than the lowest of the two weights than this method equates to the Boolean-And method since it requires both methods to issue an accept and beat the threshold. If the threshold is higher than the lowest weight though, then the decision depends totally on the biometric with the highest weight since it can meet the threshold on its own, while an accept by the smaller biometric cannot. So in the two biometrics case, decision weighting amounts to single-mode biometrics.

Table 6.3 presents the results of the intermodal experiments carried out with two different biometric modes keystroking and hand geometry. Behind each result that was obtained experimentally, the predicted result according to the models presented in Subsection 4.4 is printed between braces. To limit the number of experiments further only the top and bottom classifying techniques were chosen for each biometric.

<table>
<thead>
<tr>
<th>Hand</th>
<th>Keystroke</th>
<th>Fusion</th>
<th>FAR</th>
<th>FRR</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48</td>
<td>J48</td>
<td>And</td>
<td>2.7 % (2.8)</td>
<td>39.68 % (41.3)</td>
<td>19.6 % (20.6)</td>
</tr>
<tr>
<td>J48</td>
<td>J48</td>
<td>Or</td>
<td>23.5 % (31.7)</td>
<td>7.0 % (3.8)</td>
<td>15.9 % (18.8)</td>
</tr>
<tr>
<td>J48</td>
<td>SVM</td>
<td>And</td>
<td>3.3 % (2.5)</td>
<td>37.0 % (37.0)</td>
<td>18.8 % (18.4)</td>
</tr>
<tr>
<td>J48</td>
<td>SVM</td>
<td>Or</td>
<td>26.2 % (29.2)</td>
<td>0.8 % (1.5)</td>
<td>14.4 % (16.5)</td>
</tr>
<tr>
<td>Logistic</td>
<td>J48</td>
<td>And</td>
<td>0.7 % (1.1)</td>
<td>19.7 % (16.1)</td>
<td>9.4 % (8.0)</td>
</tr>
<tr>
<td>Logistic</td>
<td>J48</td>
<td>Or</td>
<td>25.5 % (25.2)</td>
<td>0.0 % (0.6)</td>
<td>13.8 % (13.9)</td>
</tr>
<tr>
<td>Logistic</td>
<td>SVM</td>
<td>And</td>
<td>0.0 % (1.0)</td>
<td>10.2 % (9.8)</td>
<td>4.7 % (5.1)</td>
</tr>
<tr>
<td>Logistic</td>
<td>SVM</td>
<td>Or</td>
<td>18.0 % (22.5)</td>
<td>0.8 % (0.3)</td>
<td>10.1 % (12.3)</td>
</tr>
</tbody>
</table>

Table 6.3: Results for decision level fusion with two biometrics.

The results are rather disappointing but are rather closely predicted by the models. There are some deviations but that is because the predictions treat the error rates as probabilities. Since the number of tested vectors is relatively small, there are some deviations from these predictions. Using the multimodal fusion in this way clearly
has no benefit, and the combinations have even higher error rates than when they are used separately (except for a single underlined case). This is due to the FAR and FRR rates of the keystroking and hand geometry biometrics. Looking at those rates, one can say that the keystroking biometric is rather good in verifying genuine users (low FRR) but appalling in keeping the imposters out. The hand geometry biometric is just the opposite, so using either Boolean-And or Boolean-Or brings out the worst of both biometrics. Imagine that both biometrics suffered from a high FAR but had a low FRR, then a Boolean-And fusion would have improved things. On the other hand, if the key requirement of a multimodal biometric system was to have a very low FAR to provide maximum security, then the Boolean-And method provides good results. In one case it even achieved not a single false positive in the complete test set! For this research however the goal is to minimize both the FAR and FRR in order to improve the error rate.

Since all the classifying methods of the keystroke module feature a very high FAR and a much lower FRR, fusing two different keystroke methods using the Boolean-And method might yield a system with a lower error rate. The results of this intramodal experiment are shown in Table 6.4. Again the predicted results are printed in braces behind the experimentally obtained results. Since none of the individual keystroke biometrics offer good error rates, a fusion of them should give better results. Since all the keystroke classifiers have a high FAR and a relatively low FRR, only the Boolean-And rule will be tested since the Boolean-Or yields no benefit in this case. This is because the fused FAR will be much worse than the FAR rates of the individual biometrics, giving rise to an overall error rate which is probably worse than any of the individual error rates.

<table>
<thead>
<tr>
<th>Method 1</th>
<th>Method 2</th>
<th>FAR</th>
<th>FRR</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48</td>
<td>Logistic</td>
<td>8.7 % (4.0)</td>
<td>15.3 % (16.5)</td>
<td>12.6 % (9.5)</td>
</tr>
<tr>
<td>J48</td>
<td>SVM</td>
<td>9.5 % (3.8)</td>
<td>14.0 % (14.9)</td>
<td>12.1 % (11.0)</td>
</tr>
<tr>
<td>Logistic</td>
<td>SVM</td>
<td>11.62 % (3.4)</td>
<td>8.1 % (9.8)</td>
<td>9.6 % (7.1)</td>
</tr>
</tbody>
</table>

Table 6.4: Results of fusing two keystroke classifiers using the Boolean-And rule.

For each of the 132 keystroke users the complete test set of 518 genuine and 368 imposter vectors was used. The two different classifiers were fed with the same vector and then fused. Again the results were largely as expected, achieving a slight improvement of the error rate and especially the FAR at the cost of an increase of the FRR. The deviation from the expected scores was bigger though than in the earlier experiment and this probably due to the lack of independence between the two different classifiers since each of the keystroking classifiers is trained on the same common data set. It is much more likely that the classifiers have problems with the same subset of vectors of the data set, which negate the added value of fusing them.

### 6.2.2 Fusion With Three Biometrics

For fusion with three biometrics there are more variants available requiring at least two of the same biometric modes. First, a single hand geometry biometric will be fused with two keystroke biometrics. Since the hand geometry biometrics perform better in general than the keystroking, fusing more hand biometrics than keystroking ones will likely overshadow the contribution of the keystroking. To see if the performance of a single biometric can be improved by using three different classifiers there will also be two intramodal experiments using the three classifiers of the respective modes.

#### Intermodal Fusion

Although the Boolean-And and Boolean-Or rules can be readily applied to the three mode fusion, it can be predicted that their application will not improve the overall error rate. Consider for instance the application of the Boolean-And rule to a hand biometric (J48), and two keystroke biometrics (J48 and SVM). The predicted FAR, FRR and error rates are then predicted to be:
The results of the majority voting are mixed. In the first test instance, using the J48 classifier, the results are prolific: the overall error is much lower than any of the individual techniques and also the FAR and FRR rates are better in general. This is mostly due to the very high FRR rate of the J48-based hand geometry biometric. When looking at the other two test instances it becomes clear that the better hand geometry classifier is ‘outvoted’ by the two keystroke biometrics, making the overall result worse than the individual hand geometry biometric. It is likely that since the keystroke classifiers are based on the same training data, that they will also have problems with the same subset of vectors. The ‘problem cases’ will then dominate the result since the hand biometric is outvoted.

For the decision weighting the error rates will be used to base the weights on, instead of the EER. This is because the EER was unavailable for the different biometrics. There are a number of different scenarios possible with different weights and threshold combinations. Remember that the weights sum to one and that when a biometric accepts a given vector, its weight is added to a sum. This sum is then compared against the threshold $L$ and when the sum is equal or greater than the threshold, the vector is accepted. For the three biometric modes case, assume that the weights are all different: $0 < w_1 < w_2 < w_3 < 1$. There are a number of cases to consider, of which only one is interesting for our purposes.

- $L \leq w_1$: This equates to Boolean-Or.
- $w_1 < L \leq w_2$: This is not useful. The least reliable biometric cannot force an accept or reject and is not relevant in the eventual decision. This amounts to a two Boolean-Or rule on the two best biometrics.
- $w_2 < L \leq w_3$: This is useful. Either the best biometric accepts the user or both the other two have to do it.
- $w_1 + w_2 < L$, $w_2 + w_3 < L$ or $w_1 + w_3 < L$: Useless because the decision depends totally on $w_3$, $w_1$ and $w_2$ respectively.

<table>
<thead>
<tr>
<th>Hand</th>
<th>Keyst. 1</th>
<th>Keyst. 2</th>
<th>FAR</th>
<th>FRR</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48</td>
<td>Logistic</td>
<td>SVM</td>
<td>15.2% (8.4)</td>
<td>3.9% (3.1)</td>
<td>9.4% (6.0)</td>
</tr>
<tr>
<td>Bayes</td>
<td>Logistic</td>
<td>SVM</td>
<td>14.8% (7.0)</td>
<td>2.5% (1.0)</td>
<td>9.7% (4.2)</td>
</tr>
<tr>
<td>Logistic</td>
<td>Logistic</td>
<td>SVM</td>
<td>13.7% (5.5)</td>
<td>1.8% (0.7)</td>
<td>7.9% (3.3)</td>
</tr>
</tbody>
</table>

Table 6.5: Results of fusing one hand and two keystroke biometrics using majority voting.
• $w_3 < L$: This variant can be useful given that none of the conditions of the previous variant hold. In that case, this variant equates to the majority voting method with a required majority of 2.

Table 6.6 gives the results for the experimentation. Again, the results are not encouraging since in all cases the overall error rate is similar or higher than the individual error rate. This is very likely caused by the two keystroking classifiers which make the same misclassifications in the same places and thereby dominating the decision making, similar to the majority voting case.

<table>
<thead>
<tr>
<th>Hand</th>
<th>Keystroke1</th>
<th>Keystroke2</th>
<th>FAR</th>
<th>FRR</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>J48</td>
<td>Logistic</td>
<td>SVM</td>
<td>32.6%</td>
<td>0.0%</td>
<td>17.6%</td>
</tr>
<tr>
<td>Bayes</td>
<td>Logistic</td>
<td>SVM</td>
<td>36.0%</td>
<td>0.8%</td>
<td>20.4%</td>
</tr>
<tr>
<td>Logistic</td>
<td>Logistic</td>
<td>SVM</td>
<td>38.7%</td>
<td>1.6%</td>
<td>21.6%</td>
</tr>
</tbody>
</table>

Table 6.6: Results of fusing one hand and two keystroke biometrics using decision weighting.

Intramodal fusion, or the fusion of three different classifiers for a single biometric, was not attempted since by now it became clear that the different classifiers are highly dependent. Therefore it is not expected that a straightforward fusion will improve the error rates. Unfortunately the other keystroking methods, such as the rhythm-based or digraph-based approaches, were not included in the keystroking module in the first place because of their high error rates. Because these techniques are based on different data, it can be assumed that they are more independent of the current keystroke classifiers. A fusion with those other techniques might yield better results for the keystroke verification problem.

### 6.3 Match Level Fusion Experiments

Before presenting the results of the multimodal fusion on the match or score level a number of reservations have to be made. As said before, all the biometric modules use Weka classes for their classification. Each of those Classifier classes implements the `distributionForInstance` method which returns the predicted class memberships for a given test instance. This return value takes the form of an array of doubles with values between 0 and 1. Because it was very difficult to obtain the real scores obtained from the classifiers, these predicted class memberships were interpreted as the score of the instance under test.

For instance, consider a binary classifier as used with the keystroking module for each individual user. During verification, a test vector is passed to the classifier which predicts class memberships of the two classes: ‘genuine’ and ‘imposter’. The score obtained is taken as the probability associated with the ‘genuine’ class. For a multi-class classifier as used on the hand geometry data, the procedure is almost the same except that for this classifier the predicted class with the highest probability is used. If the predicted class is a user, the probability is returned as the score. If the predicted class with the highest probability $p$ is the ‘unknown’ class though, then $1-p$ is returned as the score. This is done to make the hand geometry scores compatible with the keystroking scores: a higher score means the more likely it is that we’re dealing with a genuine user.

The problem however with using the `distributionForInstance` method to compute the scores is that it is not really a score. In most cases, the Weka classifier has already applied some thresholding to come up with these class predictions. This is apparent for example in the support vector machines and J48 classifiers: all the predicted class memberships produced by these classifiers are either 0 or 1. This makes the separation of the genuine and imposter distributions by means of the QQ and QLQ normalizations useless. Furthermore, the main advantage of using match level fusion falls away since having only a 0 or 1 score is essentially the same as having an ‘reject’ or ‘accept’ decision. Fortunately, there were also classifiers that exhibited a little more variation in the scores. These are the logistic regression classifier for the keystroking (Figure 6.2) and the naive Bayes classifier for the hand geometry (Figure 6.3). In the case of the logistic regression there is little variability but the naive Bayes classifier shows more variability in the scores. It becomes clear though that both distributions are heavily concentrated at the extremes and that there is considerable overlap between the genuine and imposter distributions at these extremes.
The figures also show that using a technique like QQ or QLQ will not yield any benefits since there is a big overlap of the scores and the distribution is quite different than that depicted in Figure 4.2. Therefore no normalization will be applied to the scores since they are also in the appropriate $[0, 1]$ score range. Only the earlier mentioned classifiers will be used for the experimentation at this level. Also, only fusion between two different biometric modes will be attempted to avoid the dependence problem that became apparent when fusing with only one mode. The experiments were done in a similar way as the intermodal decision level fusion which means that only a subset of all the keystroke users could be used (those who are also enrolled into the hand geometry biometric). The results for the four different fusion methods (simple-sum, min-score, max-score and matcher weighting) are depicted in Figures 6.4, 6.5, 6.6 and 6.6 respectively. Note the $[0, 2]$ score range for the simple sum method.

The best results of an error rate of 2.7 percent was obtained with the matcher weighting method, followed by the simple sum at 4.1 percent. The min-score method was slightly worse at 4.7 percent. The max-score method was the worst with 8.6 percent, only slightly better than the individual classifiers. In the original distributions there was a significant number of imposters that achieved the maximum score. Since the biometrics are independent, the chances that both classifiers give a maximum score for a given imposter are much smaller, which was capitalized upon by the simple-sum and matcher weighting methods.

Both these methods sum the scores, either directly or after having applied weights. This tends to separate the genuine and imposter distributions since they are spread over a bigger scoring range, given independence between the biometrics that are fused. The humps that can be seen in the graphs of the simple-sum and the matcher weighting methods in the middle are caused by an imposter that is highly classified by one classifier but very low by the other. The imposters that achieve a high score for both classifiers get higher scores but their numbers are very few. Furthermore, the FRR rates of both the hand geometry as well as the keystroke classifers are lower than their respective FAR rates so the genuine scores are mostly distributed over the upper half of the scores. This makes for good thresholding and separation of the genuine and imposter distributions which explains the success of the simple sum and matcher weighting methods. The matcher weighting method gets the edge over the simple sum because of its bias for the best classifier (in terms of error rate) involved in the fusion.

The min- and max-score are not able to able to separate the genuine and imposter distributions in the way summing does and therefore have lower performances. The min-score takes the lowest score and is therefore biased towards rejection. This explains the improved error rate over the individual biometrics since it downplays the very high FAR rate of the keystroke biometrics. The max-score method tends towards acceptance and therefore runs into problems with high FAR rate of the keystroking. The better results obtained with simple-sum and matcher weighting over min-score are consistent with [Ind03].

All match level fusion methods represent an improvement over the individual techniques. This is because they are based on the scores and their variability, instead of the black-and-white decisions. These variabilities in the scores allow for more accurate thresholding and fine-tuning which accounts for the better error rates.
6.4 Overall Results

This section will repeat the best results obtained for the different fusion variants that were attempted and the improvements made by applying those variants. Table 6.7 gives the best results that were achieved for each method. The FAR, FRR, and error rate are given for each of the individual biometric involved and for the combined system after multimodal fusion was applied. The rates are given as percentages with the FAR rate first, followed by the FRR and overall error rate. This table clearly shows the contrast between the decision level fusion and match level fusion.
<table>
<thead>
<tr>
<th>Fusion Variant</th>
<th>Rates 1</th>
<th>Rates 2</th>
<th>Rate 3</th>
<th>Fused Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision level (Boolean-And), keystroking + hand</td>
<td>20.1, 4.9, 11.2</td>
<td>17.9, 3.8, 9.7</td>
<td>0.0, 10.2, 5.1</td>
<td></td>
</tr>
<tr>
<td>Decision level (Boolean-And), keystroking + keystroking</td>
<td>20.1, 4.9, 11.2</td>
<td>17.9, 3.8, 9.7</td>
<td>11.62, 8.1, 9.6</td>
<td></td>
</tr>
<tr>
<td>Decision level (Majority), 2 x keystroking + hand</td>
<td>5.4, 5.7, 5.5</td>
<td>20.1, 4.9, 11.2</td>
<td>17.9, 3.8, 9.7</td>
<td>13.7, 1.8, 7.9</td>
</tr>
<tr>
<td>Decision level (Weighting), 2 x keystroking + hand</td>
<td>5.4, 5.7, 5.5</td>
<td>20.1, 4.9, 11.2</td>
<td>17.9, 3.8, 9.7</td>
<td>38.7, 1.6, 21.6</td>
</tr>
<tr>
<td>Match level (Simple-Sum), keystroking + hand</td>
<td>20.1, 4.9, 11.2</td>
<td>17.9, 3.8, 9.7</td>
<td>4.7, 3.4, 4.1</td>
<td></td>
</tr>
<tr>
<td>Match level (Min-Score), keystroking + hand</td>
<td>20.1, 4.9, 11.2</td>
<td>17.9, 3.8, 9.7</td>
<td>5.5, 3.3, 4.5</td>
<td></td>
</tr>
<tr>
<td>Match level (Max-Score), keystroking + hand</td>
<td>20.1, 4.9, 11.2</td>
<td>17.9, 3.8, 9.7</td>
<td>0.0, 18.7, 8.6</td>
<td></td>
</tr>
<tr>
<td>Match level (Weighting), keystroking + hand</td>
<td>20.1, 4.9, 11.2</td>
<td>17.9, 3.8, 9.7</td>
<td>1.6, 4.0, 2.7</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.7: Best results of different fusion variants.
Chapter 7

Conclusions and Recommendations

7.1 Conclusions

The Bio-SPHERE platform, which was developed during the course of the project lives up to its requirements. Its architecture and design, although not very complicated, gets the job done: using Bio-SPHERE it is possible to integrate different biometrics and do multimodal experiments with them due to the flexible service-oriented architecture. The platform fulfills its task with very little overhead and offers its services in a simple and effective manner. Since the platform uses common protocols such as WSDL and HTTP it is also quite easy to interface with. During the project, C- and Python-based libraries were used to communicate with the platform, but any language that has facilities for XML and HTTP can be adapted to make use of Bio-SPHERE ’s services. In its current state, it is even possible to use the biometrics services as a web service, for instance to protect a website with keystroking biometrics.

Also the concept of modules, as implemented in the platform, allows the developers of biometrics and other services great freedom. The author experimented for instance with biometrics based on a mix of C and Java code which worked quite well during experimentation. The modules are free to open files, network sockets and databases or to create a fork and load other programs. The addition of the biometric’s services requires in most cases only the creation of a wrapper layer to correctly handle services and responses. The code is also meticulously documented so it should be easy for researchers and other users to extend the platform.

During the experimentation with the keystroking data the shortcomings of the keystroking database became clear. The biggest problem was that for each user only a little keystroking information was available; only his or her name. This gave problems for the application of commonly used techniques such as the digraph-based approach. Since all the users typed a different string (their name), there were no features common to all users. It is also typical in biometrics that only a small portion of the data set is used: the portion that lends itself for the best classification. In the case of keystroking, such a selection was not possible since the number of keystroke events for each user was already very low. In other researches into keystroking a user is typically required to type a small text which yields sufficiently big data sets. From the experimentation it became clear that a user-dependent approach was the best approach for the given database. The advantages of a user-dependent approach is that is specially trained for a particular user and is also scalable: the addition of a new user does not require a retraining of the complete model.

The results of the multimodal biometrics were mixed. Decision level fusion offered only slight improvement in some cases but was actually worse than the individual biometrics in quite a number of cases. This was because of the ’reversal’ of the FAR and FRR rates between the keystroking and hand geometry biometrics: matching a system with a high FAR and low FRR rates between the keystroking and hand geometry biometrics. Intramodal fusion gave a slightly better system, but because of the high dependency between the intramodal classifiers used the results were not as good as expected. Match level proved to be much more powerful than decision level fusion. It offered significant improvements in the error rates for almost all of the applied methods. The method based on summing the individual scores (Simple-Sum and Matcher Weighting) offered the best biometric performance, since they were able to achieve the greatest separation of the genuine and imposter populations.
In all the experiments, the Weka data mining platform proved itself to be a very quick and easy way to investigate and classify data. Its very elaborate set of stand-alone classifiers was used successfully in the two biometric modules. However the classifiers only prove themselves useful at the decision level. Weka does not provide the raw scores but only provides a slight indication through the distributionForInstance method. Its classifiers also offer only limited tuning facilities. Particularly for match level fusion it is very probable that better results can be obtained with the real scores and other classifiers.

### 7.2 Recommendations

To keep the platform from becoming too complicated, the HTTP 1.0 protocol was implemented which creates a new connection for each request. When a client tries to make numerous requests, this results in a considerable overhead because of the continuous connection setup and is therefore inefficient. This can become a bottleneck when having computationally limited devices or slow networks. HTTP 1.1 can reuse a socket and so it can reduce the overhead of clients that make numerous requests, such as when adding a lot of training vector to a biometric. If the platform is to be used outside of research it should be made more secure by using the more secure HTTPS protocol instead of the plaintext HTTP.

During the design, the choice was made for to use the WSDL 1.1 standard to describe the offered services. This standard was superseded during development by the new 2.0 version, which is endorsed by the W3C. The new version is more compact and powerful and does way with the three-level hierarchy of the services, which seemed a bit overkill to describe just a handful of services. The new version also allows for documentation elements for the different services, which is very useful for potential user of the services. By adopting the new standard the platform can interoperate with other systems.

For the multimodal experimentation only a handful of user-dependent classifiers were incorporated into the keystroking module. Since these were trained on the same data set there was a great dependency between these classifiers which hampered the intramodal fusion. By fusing different keystroking methods, such as one user-dependent approach together with a rhythm and time-ranking approach, it is very likely that better results can be obtained for the keystroking verification problem because of the improved independency. Considerable effort was put into the improvement of the single keystroke classifiers but the lowest error rate is still about five percent. By using intramodal fusion it is likely that lower error rates can be achieved.
Bibliography


Appendix A

Bio-SPHERE Installation Manual

This chapter will discuss the contents of the Bio-SPHERE source code distribution and also highlight the most important portions of the code, especially the mechanism for handling service requests. Future developers can use this appendix to get themselves familiar with the platform and its internals.

A.0.1 Package Layout

Bio-SPHERE is distributed as a bzipped tarball and can be downloaded from [http://www.sf.net/projects/biosphere](http://www.sf.net/projects/biosphere). After unpacking this archive a directory structure is obtained which is listed in Listing A.1. Some less important directories and files are omitted.

```
biosphere-0.0.2/
  common/         (Common code)
  daemon/         (Server code)
  docs/           (Documentation)
  include/        (Common header files)
  library/        (Client library)
  modules/        (Modules)
    fusion/       (Multimodal fusion)
    keystroke/    (Keystroke biometrics)
    password/     (Password module)
    stat/         (Statistics & timing)
  python/         (Python-based GUI)
  tests/          (Unit tests)
  configure       (Configure script)
```

**List of Listings A.1: Contents of Bio-SPHERE source archive**

Especially the `daemon/` directory is the primary directory since it houses the server-specific code. The `common/` directory contains code that is used by both the server and its client library, which is located in `library/`. It is compiled into a convenience library and then linked against both the server and the client library. The source code for the modules is contained in separate subdirectories under the `modules/` directory.

In the root of the package there is also a standard `configure` script which is used for building and installing it. It follows the standard configure-make-make install adagium, so for example

```
tar xvzf biosphere-0.0.2.tar.bz2
cd biosphere-0.0.2
./configure --prefix=~/biosphere
make
```
make install

will install Bio-SPHERE under the user’s home directory.

During the building process the server (biosphered), the client library (libbiosphere.so) and the various modules (bsmod_* .so are built. These are placed in during installation is standard directories listed in Listing A.2

biosphere/
  bin/ (Binary directory)
    biosphered (Server binary)
    core_services.wsdl (Core WSDL definitions)
  etc/ (Configuration directory)
    biosphered.conf (Server configuration file)
  include/ (Header file directory)
    biosphere.h (Global definitions)
    biosphere_error.h (Error codes)
    biosphere_lib.h (Library header file)
    biosphere_module.h (Module header file)
  lib/ (Client library directory)
  modules/ (Modules directory)

List of Listings A.2: Contents of Bio-SPHERE installation directory

All the executable binaries of the Bio-SPHERE platform are installed in the bin / directory under the installation prefix. This includes the server and its accompanying WSDL file. The etc/ directory contains a configuration file that is read by the server on startup, setting various options. The library is located in the lib/ directory and all the modules are in the modules directory. Note that, in contrast with the source package, modules do not have their own subdirectory there but instead reside side-by-side. Their extra data files are also stored in this directory.

A.0.2 Running Bio-SPHERE

After Bio-SPHERE is fully installed it can be started by issuing the following command:

```
/usr/local/bin/biosphered > /var/tmp/biosphered.log &
```

This will put the server in the background and have it output to the specified logfile. It is now ready to service clients. In addition a number of extra commandline options can be passed as well:

- `-m <module name>`. Module name must be the name of the dynamic library and by asserting this option, Bio-SPHERE automatically loads the specified module on startup.
- `-n <node address>`. This way the server becomes aware of another node. The node address can be an IP address including the port or a DNS name. The port number is optional.
- `-i <limit>`. The maximum number of concurrent connections that is accepted.
- `-p <portnumber>`. The TCP port to start listening on for incoming connections.
Appendix B

Bio-SPHERE Library Programming

B.1 Introduction

CLIENTS of the Bio-SPHERE platform make use of its services through the Bio-SPHERE client library. This library provides an easy interface for making service requests without the need to bother with the translation of that request into XML or talking HTTP with the server. The library offers basically two functions:

• **Retrieve Service Definitions.** When the user first makes contact with the server he or she probably has little awareness of the services it offers. The library offers the get_services_list function which returns both the core services as well as the modules services of the Bio-SPHERE node contacted.

• **Make a Service Request.** After the user has determined which service he or she wishes to use, the make_service_request function of the library can be called to forward the request to the server and return the corresponding request.

A simplified version of the biosphere_lib.h file, which is to be included by every program that makes use of the library, is given in Listing B.1. Before making any request or retrieving any service definition, the library should be initialized by calling init_library (line [5]). At shutdown, the library’s cleanup function cleanup_library (line [8]) should be called to make a clean exit.

The get_services_list function will fill a bs_definition structure with the definitions of all the services offered by the server. The actual requests are made using the make_service_request function. Notice that only a local server can be contacted with the library by specifying its port. Since the library makes use of the Apache Portable Runtime internally and thus allocated from memory pools it offers some special functions to make deep, malloc-ed copies of the data structures it uses in its operations. This way, the user does not have to have any knowledge of the Apache Portable Runtime.
```c
#include <biosphere.h>
#include <biosphere_error.h>

/* Initialize the library: */
bs_status init_library(void);

/* Cleanup the library: */
bs_status cleanup_library(void);

/* Retrieve all service definitions from server: */
bs_status get_services_list(bs_definition **def, bs_uint16 port);

/* Make a service request: */
bs_status make_service_request(bs_service_request *request,
                                bs_service_response **response, bs_uint16 port);

/* Make deep copy using malloc(): */
bs_definition *copy_bs_definition(const bs_definition *def);

/* Free definition structure: */
void delete_bs_definition(bs_definition *def);

/* Create deep copy of response using malloc(): */
bs_service_response *copy_bs_service_response(
                                const bs_service_response *response);
```

List of Listings B.1: Simplified header file `biosphere_lib.c`

### B.2 Example Program: Shutdown

To illustrate the use of the library a simple program called `shutdown` will be created. The program’s task is to shut down the local Bio-SPHERE node by calling the Shutdown core service. The user of the `shutdown` program can specify a timeout on the command line before the server really shuts down. The Shutdown message is part of the core services and is covered in the excerpt of the cores services WSDL definition:
Next, the actual code for the shutdown program will be presented. The first part of the code contains the inclusion of the necessary header files, most importantly biosphere_lib.h. The program consists of a single main function. On lines [8], [11] it declares the necessary data structures for making a service request and then proceeds by initializing the library and parsing an optional user-given timeout (line [15]).

```c
#include <biosphere_lib.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int timeout = 0, rv;
    bs_data_type *type; /* For the bs_uint32 type */
    bs_message_instance *msg; /* The Shutdown msg */
    bs_message_part *part; /* The timeout part */
    bs_service_request *req; /* Service request */
    bs_service_response *resp; /* Response */
    init_library();
    if (argc > 1) timeout = atoi(argv[1]);
```

After initialization the service request must be created. Since a bs_service_request is a moderately complex data structure, it is built in steps. First, the data type used (bs_uint32) and the message part are created in the following code listing. Again calloc is used extensively to clear the data structure before using it. Since the bs_uint32 data type is a built-in type, the built-in flag of the data type is set to true. The specified timeout is converted to a string and will be the data of the message part.

```c
    /* Create data type: */
    type = (bs_data_type *) calloc(1,
```
```c
  sizeof(bs_data_type));
type->builtin = TRUE;
type->name = (char *) malloc(strlen("bs_uint32") + 1);
strcpy(type->name, "bs_uint32");

/* Create message part: */
part = (bs_message_part *) calloc(1,
  sizeof(bs_message_part));
part->name = (char *) malloc(strlen("timeout") + 1);
strcpy(part->name, "timeout");
part->type = type;
part->data = (char *) calloc(1, 10);
sprintf(part->data, "%d", timeout);
part->size = strlen(part->data);

After the message part has been created, the ShutdownMessage can be allocated (line 33) and instantiated.
Notice the use of calloc to allocate the parts pointer array. Its first parameter is the number of data structures to
allocate, in this case only one.

  /* Create Shutdown Message: */
msg = (bs_message_instance *) calloc(1,
  sizeof(bs_message_instance));
msg->name = (char *) malloc(strlen("ShutdownMessage") + 1);
strcpy(msg->name, "ShutdownMessage");
msg->parts = (bs_message_part **) calloc(1,
  sizeof(bs_message_part *));
msg->parts[0] = part;

Finally the request itself is built using the earlier created ShutdownMessage which will serve as the input
message. Then the actual service request is made, using the standard 2915 port. Afterwards, the response will
be contained in resp but since the goal of the program is to shutdown the server the response is not used. The
returned status code of the request is used as the return value of the program.

  /* Create the shutdown request: */
req = (bs_service_request *) calloc(1,
  sizeof(bs_service_request));
req->service = (char *) malloc(strlen("BiosphereCoreServices") + 1);
strcpy(req->service, "BiosphereCoreServices");
req->port = (char *) malloc(strlen("DaemonAdministration") + 1);
strcpy(req->port, "DaemonAdministration");
req->operation = (char *) malloc(strlen("Shutdown") + 1);
strcpy(req->operation, "Shutdown");
req->input = msg;

  /* Make request to server (ignore response): */
rv = make_service_request(req, &resp, 2915);
cleanup_library();
  return rv;
}
```

It should be noted that the previously presented program is devoid of any bounds checking or error handling.
In real-life applications these should be present, but they were omitted for the sake of simplicity. The shutdown
program can be compiled with the following gcc command:
Besides a client library based on C, there is also a simple library written in Python. This library features more or less the same functionality as the C library and can be used to let Python programs request services from Bio-SPHERE. The Python-based graphical front-end makes use of this library’s functionality. This library was used extensively during the experimentation phase since python allows for quick and flexible prototyping. All code is contained in a Python module called biosphere which contains a number of submodules listed in Listing B.4:

```
biosphere/
    io.py    # Contains main communication class
    types.py # Equivalents of \biosphere types & structures
    xlate.py # XML translation engine
```

**List of Listings B.4: Contents of the Python module**

The main class used by the library is the BiosphereConnector class which can connect to a Bio-SPHERE server, retrieve its service definitions and make requests to it. A simple program listed in Listing B.5 illustrates the use of this class and the library in general. As can be clearly seen from this simple program is that Python saves the programmer a lot of allocation and other hassles.

```python
import biosphere.io
import biosphere.types
conn = BiosphereConnector()

# create a shutdown request to the Bio-SPHERE server:
int_type = bs_data_type('bs_uint32')
input = bs_message_instance('ShutdownMessage')
timeout_part = bs_part_instance('timeout', int_type, '0')
input.parts.append(part)
request = bs_service_request('BiosphereCoreServices',
    'DaemonAdministration', 'Shutdown')

# make service request; response can be checked for errors:
response = conn.RequestService(request)
if not response: print 'There was an error!', conn.error
conn.close
```

**List of Listings B.5: Example program using the Python module**
Appendix C

Bio-SPHERE Module Programming

The principal way to extend the Bio-SPHERE platform with new services is to write a module for it. Modules are dynamically loadable libraries that contain the functionality to handle additional service requests. Modules can be loaded and unloaded on the fly, can be written in C or C++ and are relatively straightforward to create. There is a great amount of freedom allowed when writing a module. Modules run under the same privileges as the server and can contain and execute any code their makers put into them. This brings a lot of responsibility on module writers since they can also execute all systems calls, but also ensure maximum freedom and flexibility for the module’s implementor.

A Bio-SPHERE module always consists of the following parts:

- **Dynamic Library.** This library can be dynamically loaded and unloaded by the Bio-SPHERE platform and contains the executable code that handles the service requests meant for the module. The library also declares a global variable called bsmod_symtable that defines various attributes and function pointers.

- **Definition File.** This is a WSDL file describing the services offered by the module, which will be used by Bio-SPHERE to determine which services are offered.

- **Optional Data Files.** Some modules use for instance a database file or other types of files to assist them in their servicing.

Modules should be installed in the modules/ subdirectory under Bio-SPHERE ’s installation root in order for the platform to find them. By default this amounts to /usr/local/biosphere/ on Linux and Mac OS X. In order to illustrate how these different parts come together a simple tutorial is presented next.

C.1 The Quote Service

To illustrate the creation of a module an example will be used. In this case, a user wishes to extend Bio-SPHERE by adding a new service called quote. This service allows users to post nice quotes they have found which will be appended to a file called quotes.txt. A quote consists of the quote’s author and of course the message itself.

C.1.1 Service Definitions of the Quote Module

The first step in creating this quote module, which will be called bsmod_quote, is to define the services it will offer. This is done by writing the WSDL file which is listed in Listing C.1.

As can be seen on line 2, the bsmod_quote module uses only a type, namely the built-in bs_string. By using a built-in type, Bio-SPHERE will take care of type and bounds checking. If the module uses custom types, they will have to be declared here and possible type checked by the module itself.

The module uses two different kinds of messages. The QuoteMessage on line 4 is used for packing the quote data and providing it as an input to the quote service. It consists just of two parts: the author of the quote and the quote itself. The ReturnMessage does not have any parts and is send back to the user in the service response to signal the addition of the new quote.
There is only a single port type for this limited module named QuotePortType (line 10). This port type defines a single operation SaveQuote which is used to save the quote. The quote and its author are provided in a QuoteMessage and this operation signals its completion by sending back a ReturnMessage.

The only service defined by the module is called BSModServices and is defined on line 17. The 'BSMod' prefix is a convention to distinguish module offered services from core services. There is only a single port in the service called QuotePort which is of the earlier defined QuotePortType. Having the complete service-port-operation hierarchy is probably overkill for such a simple module but makes much more sense when multiple services are involved.

### C.1.2 Creating the Module Skeleton

After the services of the Quote module have been clearly defined, a basic service handling infrastructure must be created for it. Like all modules, the bsmod_quote module must define a structure called bsmod_symtable of type bs_module. This structure is defined in biosphere_module.h.

For the Quote module a single C file (bsmod_quote.c) will suffice to offer all the functionality. Listing C.2 lists the basic skeleton, including the bsmod_symtable structure.

At this point the module can be compiled and loaded by the Bio-SPHERE platform, although it is not actually useful. The only requirement for the compiler is the inclusion of the necessary header files, located in the include/ subdirectory of the installation. The bsmod_symtable structure defined on line 15 defines some information about the module such as its version, its author and a description. It also sets function pointers to initialization, cleanup and service handling routines.

The initialization routine at line 4 is called immediately after the module is loaded and is used to initialize the module. What entails is up to the module writer, but in the case of the Quote module this involves the opening of the quotes.txt file. In a similar manner, the cleanup function is called before unloading the module (line 7).

The bsmod_quote_handle function on line 10 that is yet to be defined, will perform the actual service handling. It takes a bs_service_request structure detailing the request received from the user, acts upon it, and creates a corresponding bs_service_response structure. In the case of the Quote module this is rather
/* bs_module and other types and constants: */
#include <biosphere_module.h>

static bs_status bsmod_quote_init(void)
{ return BS_OK; }

static bs_status bsmod_quote_cleanup(void)
{ return BS_OK; }

static bs_status bsmod_quote_handle(
    const bs_service_request *request,
    bs_service_response **response, void *extra)
{ return BS_OK; }

bs_module bsmod_symtable {
    0, 0, /* Major/minor version of biosphere API. */
    0, 1, /* Major/minor version of this module. */
    "bsmod_quote", /* Module's name. */
    "M.A.Hartman", /* Author name(s). */
    "june 1, 2007", /* Date */
    "(C) 2007, M.A.Hartman.", /* License info */
    /* Short description of the module: */
    "A module which stores quotes submitted by users.",
    bsmod_quote_init, /* Init routine. */
    bsmod_quote_cleanup, /* Cleanup routine. */
    bsmod_quote_handle, /* Service handling routine: */
    NULL, /* Will be set by platform. */
    "bsmod_quote.wsdl" /* Path to the WSDL file. */
};

List of Listings C.2: Basic skeleton code in bsmod_quote.c
easy: there is only type of request that has to be serviced and also the response is fixed.

One last remark about concerning the extra void pointer that is passed to the handler function. Internally, Bio-SPHERE uses memory pool allocation since it is built on top of the Apache Portable Runtime. When the handler function of a module is called, a memory pool is passed through the extra argument. The implementor of the module has the choice to allocate from this pool or ignore it altogether and use traditional memory allocation through malloc and equivalents. The Quote module will use conventional memory allocation.

C.1.3 The Initialization and Shutdown Functions

After creating a skeleton for the Quote module the initialization and shutdown functions can be implemented as shown in Listing C.3 (only the relevant functions are shown).

```c
#include <stdio.h>

FILE *quote_file; /* File pointer to quote file */

static bs_status bsmod_quote_init(void)
{
    quote_file = fopen("quotes.txt", "a");
    if (quote_file == NULL) return BS_ERROR;
    else return BS_OK;
}

static bs_status bsmod_quote_cleanup(void)
{
    fclose(quote_file);
    return BS_OK;
}
```

List of Listings C.3: The initialization and cleanup functions in bsmod_quote.c

For such a simple module the initialization consists of only opening the quotes.txt file on startup and closing it on exit. Not that all the modules are unloaded automatically when the Bio-SPHERE platform shuts down.

C.1.4 The Service Handler Function

The actual handling of new quote service request is done by the service handler function bsmod_quote_handle. This function has to extract the author and quote data from the passed QuoteMessage, append it to the quotes.txt file and create a ReturnMessage. The full code is given in Listing C.4. Note that the code in the listing builds upon the code from Listings C.2 and C.3. The code is also simplified and crucial parts error and bounds checks on the given data are omitted for clarity.

The function handler starts by taking the author and quote data and outputting it to the quote file on line [10]. This is of course rather dangerous in such a direct manner: real-world modules should carefully check the passed request structure, especially when defining custom data types which are not type checked by Bio-SPHERE.

After the request has been handled the rest of the code is dedicated to allocating an appropriate response. On line [15] a bs_message_instance structure is allocated for the ReturnMessage. Notice the use of the calloc function which also clears the structure, effectively setting a number of attributes to their appropriate values. Most notably the number of parts in the message and a boolean attribute indicating whether this structure was allocated from a memory pool (set to FALSE this way).

Line [21] allocated the main response structure and copies some attributes from the original request. It also sets the output attribute to the newly created return message. Again by using calloc, some attributes inside the
#include <stdlib.h>

static bs_status bsmod_quote_handle(
    const bs_service_request *request,
    bs_service_response **response, void *extra)
{
    bs_message_instance *ret;

    /* Retrieve and write quote to file: */
    fprintf(quote_file, "\n\n\n%sn%s\n\n", request->input->parts[0]->data,
        request->input->parts[1]->data);

    /* Create ReturnMessage (no parts): */
    ret = (bs_message_instance *) calloc(1,
        sizeof(bs_message_instance));
    ret->name = (char *) malloc(strlen("ReturnMessage") + 1);
    strcpy(ret->name, "ReturnMessage");

    /* Create response (note double pointer): */
    *response = (bs_service_response *) calloc(1,
        sizeof(bs_service_response));
    *response->service = request->service;
    *response->port = request->port;
    *response->operation = request->operation;
    *response->output = ret;

    return BS_OK;
}

List of Listings C.4: The service handler function in bsmod_quote.c
response structure are cleared. The pointer assignment of the service, port and operation strings from the request might be more robust by allocating space and making a copy of the strings.
Appendix D

Bio-SPHERE Graphical Frontend

For ease of experimentation and to test individual services, a graphical front-end (Bio-SPHERE Admin) was created using Python and its wxWidgets binding wxPython. wxWidgets aims to be a cross-platform graphical user interface library. Therefore the graphical front-end runs on any platform with Python and wxPython support. The looks of the program in Mac OS X and Ubuntu were remarkably similar. The Python program also used the python interface library to communicate with the platform. A screenshot of the application with highlighted windows and controls is illustrated in Figure D.1.

Operation proceeds in a top-down, left-to-right manner. The first thing a user will do is to select the IP address and port the Bio-SPHERE daemon will be listening on. This can be done from the file menu. In most cases this is a local address, with the default port 2915. After selecting 'Connect' from the file menu the program retrieves the WSDL service definitions from the server. All the services are listed in the leftmost listbox with the ports and operations listboxes being empty. After a service is selected, its available ports will become visible in the ports listbox. The same applies to the operations when a port is selected.

At this point the user can begin to enter the input message parts of the operation. This is done on the right side of the application. After filling in all the input fields of the parts, the user hits the Submit Request button which sends the request to the server. The server will then process the request and produce a result in the bottom section of the application. The different parts of the output message are displayed in a list-like manner.

In the case of an error during the service request handling, two things might happen. If it is a serious error, not per se related to the content of the request, an error popup will inform the user. If there was an error with the request handling and a fault message was generated, then this message (and its parts) will be displayed in the response section in red.

The front-end was not so much used during the experimentation since automated scripts got the job done much easier and faster, but was used heavenly for testing the platform and new modules. It was also used to cleanly shut down a Bio-SPHERE server by means of the shutdown core service. It is also a nice tool to get a feel for the use of the different services. The careful reader might notice the other tabs at the top of the program, which remain unimplemented.
Figure D.1: *Bio-SPHERE Admin’s main screen.*
Colophon

This manuscript was typeset by the author with the \LaTeX\ Documentation System on a Mac running Mac OS X. Text editing was done in \TeXShop using the \MiKTeX package. The illustrations and graphs were created with \Omnigraffle, \GNUPlot and the R statistics system.

The front cover illustration is a rind 1955 wood engraving and woodcut in black, brown, blue-grey and grey printed from 4 blocks by M. C. Escher entitled "Colorplate IV".

The body type is 11 point Times. Chapter and section titles are in various sizes of Adobe Helvetica-Narrow Bold. The monospace typeface used for program code is Adobe Courier.