Multimedia **A**uthoring and **M**anagement using your **E**yes and **M**ind

H2020-ICT-2014 - 644780

D4.1

Report on the middleware architecture and technical requirements

Dissemination level:	Public (PU)
Contractual date of delivery:	M7, November 30th, 2015
Actual date of delivery:	M7, November 30th, 2015
Workpackage:	WP4 Middleware for interaction through eyes and mind
Task:	T4.1 Middleware Architecture Design
Type:	Report
Approval Status:	Final
Version:	V0.6
Number of pages:	69
Filename:	D4.1_Architecture_and_Technical_Requirements_Final.docx

Abstract

The purpose of this document is to describe MAMEM's architecture covering the full spectrum of involved components, ranging from the base platform and the sensors layer, all the way to our Middleware, the Interaction SDK and the end-user applications. In particular, the document specifies the adopted technologies, the programming environment, the range of functionalities that will be provide by MAMEM, as well as the interfaces used to exchange information between the different layers. In the end, a logic view of MAMEM's architecture is presented, describing how everything can fit together to facilitate the development of applications operated through the user's eyes and mind.

co-funded by the European Union

Copyright

History

Version	Date	Reason	Revised by
v0.1 (alpha)	09/10/2015	alpha version to be checked by the consortium members and the coordinator	Walter Nistico
V0.2	06/11/2015	Intermediate version for proving soliciting the contribution from all involved partners	Walter Nistico, George Liaros, Dario Comanducci, Chandan Kumar, Spiros Nikolopoulos
V0.3	16/11/2015	Internal version for collecting follow-up contributions	Walter Nistico, George Liaros, Dario Comanducci, Chandan Kumar, Spiros Nikolopoulos
V0.4 (beta)	18/11/2015	Beta version for internal review	Walter, Nistico
V0.5 (pre-final)	27/11/2015	Version incorporate the comment from the internal reviews and circulate for final acknowledgement by the entire consoriutm	Walter, Nistico,
V0.6 (final)	30/11/2015	Proof reading and minor editing	Walter, Nistico Spiros Nikolopoulos

Author list

Organization	Name	Contact Information
SMI GmbH	Walter Nistico	walter.nistico@smi.de
CERTH	George Liaros	geoliaros@iti.gr
CERTH	Spiros Nikolopoulos	nikolopo@iti.gr
EB Neuro	Dario Comanducci	dario.comanducci@ebneuro.com
UNI KO-LD	Chandan Kumar	kumar@uni-koblenz.de

Executive Summary

This report presents the output of the efforts allocated under T4.1 of WP4, which deals with MAMEM architecture. The document covers the full spectrum of components that will be necessary to satisfy MAMEM's technical objectives.

In this respect, Section 2 addresses the considerations that have to do with the preferred operating system, the programing language and the specification of the computing platform. Section 3 elaborates on MAMEM's middleware placing particular emphasis on the synchronization between heterogeneous signals. Section 4 discusses the options that are available for acting as the basis of MAMEM's interaction SDK, as well as the preferred platform. Section 5 provides a quick overview of BCI applications based on eye-tracking and EEG signals, so as to motivate the necessity of an additional layer in the architecture dedicated for handling the communication between the back-end system and the end-user applications, described in Section 6. Finally, Section 7 presents the overview of MAMEM architecture describing how everything can fit together under a common framework. Section 8 concludes the report.

In presenting MAMEM's architecture we have adopted the following approach. Each section elaborates on a different layer of the architecture by describing the modules that implement the functionality of this layer and the interfaces that are used to communicate with the rest of the layers. Finally, in the last main section of the report we present a schematic overview of the entire architecture and describe the modules that belong in each layer, as well as the interfaces that are used to communicate with each other. In this description, we make sure to provide references to specific sections of this report (or other MAMEM deliverables) that elaborate on the details of each module and interface.

Abbreviations and Acronyms

API	Application Programming Interface
BCI	Brain Computer Interface
DoA	Description of Actions
ECG	ElectroCardioGram
EEG	ElectroEncephaloGram
ET	Eye Tracker
GSR	Galvanic Skin Response
GUI	Graphical User Interface
HCI	Human-Computer Interface
HMD	Head Mounted Display
HR	Heart Rate
HW	Hardware
LAN	Local Area Network
LPT	Line Print Terminal (used to designation a parallel port interface)
LSL	Lab Streaming Layer
MD	Muscular Disorder
OS	Operating System
PC	Personal Computer
PD	Parkinson Disease
SDK	Software Development Kit
SW	Software
TCP	Transmission Control Protocol
TTL	Transistor Transistor Logic
UDP	User Datagram Protocol

Table of Contents

List of Figures

List of Tables

1 Introduction

MAMEM's overarching goal, as defined in the DoA [5], is to integrate people with disabilities back into society by endowing them with the critical skill of managing and authoring multimedia content using eye-movements and mental commands. In reaching this goal, the concept of **Error! Reference source not found.** has been proposed that, among others, envisages the achievement of the following technical objectives:

Obj.1 – Capture, record and make available at the necessary scale, real-time and accurate information about eye-movements, brain electric signals and bio-measurements.

Obj.2 – Develop the necessary algorithms for translating this information into meaningful control that will take the form of semantic widgets.

Obj.3 – Implement a middleware sitting on top of current operating systems so as to make these semantic widgets available as elementary building blocks for implementing multimedia-related interfaces.

Obj.5 – Design, implement and evaluate a set of prototype interface applications that rely on MAMEM's middleware to execute the multimedia-related usage scenarios through the user's eyes and mind.

The goal of this report is to define the software architecture that will facilitate the achievement of the aforementioned technical objectives.

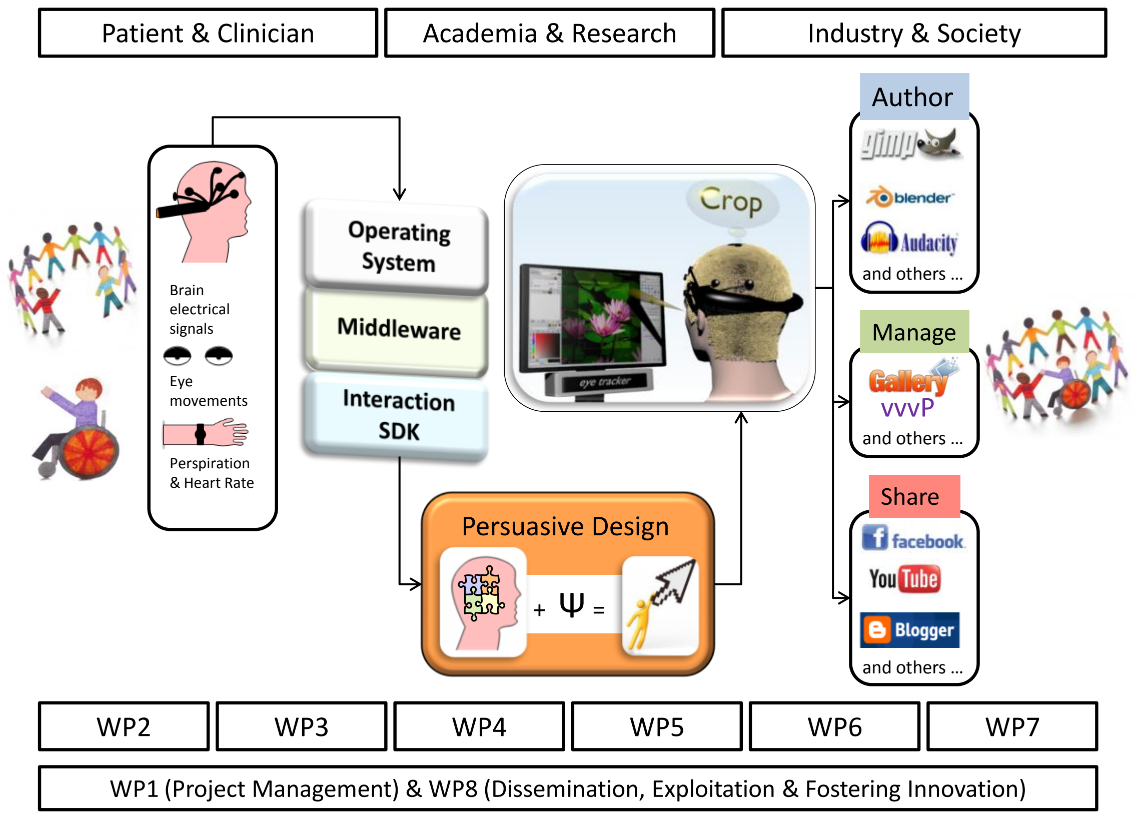


Figure 1: MAMEM concept for enabling disabled people to participate in society

What is evident from Figure 1 and the associated technical objectives is that MAMEM's architecture should be organized into layers. Indeed, Obj.1 calls for a **sensors' layer** that will acquire the signal from the sensor devices; Obj.2 calls for an **interaction layer** of algorithms and methods that will compose and SDK for HCI using eye movements and mental commands; Obj.3 calls for a **middleware layer** that will make transparent the communication between the sensor devices and the interaction layer; and finally, Obj.4 calls for an **applications layer** that will foster the development of the end-user applications. This distinction into layers was made more concrete in the description of WP4 were the arrangement of **Figure 2** was used to demonstrate the flow of information across MAMEM's software architecture. The goal of this report is to give substance in each of the boxes presented in Figure 2 and discuss the details that will be necessary to implement MAMEM's system.

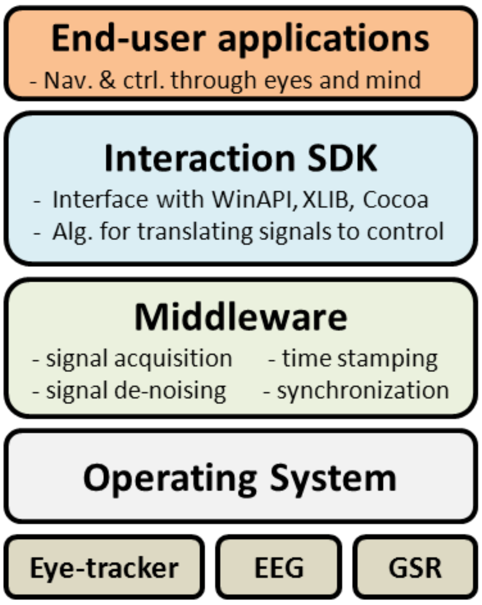


Figure 2: Different layers of MAMEM's software architecture

Apart from this preliminary identification of layers, the DoA [5] listed also a set of requirements for some of them. Given that the sensors' layer and the applications' layer were largely dependent on the technical and functional requirements - to be defined as part of WP6, more emphasis was placed on the middleware and the Interaction SDK. In the following, we summarize the goals and requirements that have been set for these layers.

Middleware: The goal of a middleware is to sit between the core API of the underlying operating system and the high-level programming environment of an SDK. Thus, in the context of BCI systems, its goal is to facilitate, on the one hand, the integration of add-on sensor modules (i.e., eye tracker, EEG recorder and bio-sensors) and, on the other hand, the execution of algorithms processing the captured signals. In what refers to the middleware, there have been four basic requirements that motivated our decisions in designing its architecture: a) hardware agnostic, in the sense of being able to support a long list of existing sensor devices (i.e. EEG, eye-tracking and galvanic skin response) and keeping the barrier very low for incorporating new devices, b) cross-platform, in the sense of being able to support all major operating systems (i.e. Windows, Apple OS and Linux), c) synchronization ready, in the sense of being able to receive signals from multiple sensors and serve them in the synchronized fashion, d) communication ready, in the sense of being able to transparently communicate with MAMEM's interaction SDK.

Interaction SDK: The goal of the Interaction SDK has been set on implementing the necessary algorithms and methods for translating the acquired signals into meaningful commands for the human-computer interface. In this case, the requirements motivating our decisions are: a) transparent, in terms of the communication with the middleware, b) rich, in terms of the supported methods and process, c) extendable, in terms of adding more methods and processes, and d) easy to use, in the sense of allowing even non-experts to "program" their own analysis process.

In presenting MAMEM's software architecture, we specify all different components involved, ranging from the sensor devices and the middleware, all the way to the interaction SDK and the communication with the end-user applications. The logic instruments that we use to describe our architecture are *Layers*, *Modules* and *Interfaces. Layers* are used to denote the parts of our system that serve a different purpose. *Modules* are used to describe the core functionalities performed in each layer and the *Interfaces* are used to specify how the information flows from one layer to another. Throughout the document we make sure to provide elaborated descriptions for each one of these *Modules* and *Interfaces* and towards the end of this report we present how the different components fit together under a common architecture.

In this respect, Section 2 addresses the considerations that have to do with the preferred operating system, the programing language and the specification of the computing platform. Section 3 elaborates on MAMEM's middleware placing particular emphasis on the synchronization between heterogeneous signals. Section 4 discusses the options that are available for acting as the basis of MAMEM's interaction SDK, as well as the preferred platform. Section 5 provides a quick overview of BCI applications based on eye-tracking and EEG signals, so as to motivate the necessity of an additional layer in the architecture dedicated for handling the communication between the back-end system and the end-user applications, described in Section 6. Finally, Section 7 presents the logic overview of MAMEM architecture describing how everything can fit together under a common framework. Section 8 concludes the report. Given that the sensors' layer is the topic of another deliverable (i.e. D2.1 – Prototype modules implementation for signal capturing [36]) it is not thoroughly discussed in this report.

Finally, it is important to note that in deciding about MAMEM's software architecture our intention has been to make the best possible use of the existing knowledge and developments in brain computer interfaces and avoid replicating the development effort that has been already undertaken by the respective community. In this respect, the open source frameworks of LabStreamingLayer [12] and OpenViBE [10] have been chosen to play an essential role in the implementation of MAMEM's system.

2 Base Platform Considerations

2.1 Operating system

In the project DoA (Description of Actions) [5], it has been set the ambitious goal to realize a platform independent architecture for the software (SW) side of the MAMEM project. While such goal has its merits to ensure the broadest possible accessibility to the system, it poses extremely high challenges in the realization of the project, for a modest return in terms of target population.

In fact according to recent statistics [1] over Desktop and Laptop internet browsing, Windows XP and subsequent Windows versions accounts for 88.97% of all web market share. If we consider tablets and smartphones the picture changes considerably, with Windows accounting for 50.83%, Android for 25.63% and iOS for 16.24%. It is however to be considered that there are no Eye Tracking devices which can be connected to smartphones, and Android tablet support it is so far only announced but not available. The situation is similar for scientific grade EEG sensors, while Emotiv's EPOC recently added support for Android and iOS [2].

Furthermore, different Operating Systems offer radically different interaction paradigms on the application side, which increases the development effort that is necessary to support all of them. So, while the proposed architecture will be flexible enough as to allow easy porting to additional operating systems as these become viable options on the sensor side, the initial efforts will be focused on getting the system to work on the platform which offers the largest coverage in terms of user base as well as available sensors, Microsoft Windows.

2.2 Programming language

We tried to identify the most convenient programming language to implement MAMEM software architecture. The main decision criteria for choosing the programming language for the realization of the system are:

1. Ease of integration with the SDKs of the sensors which will be part of the system.
2. Good performance and low resource utilization, to allow near real-time interaction and synchronization on affordable host systems.
3. Widespread knowledge and use of the programming language among the developers community, to broaden the potential developer community and leverage as much as possible existing libraries and tools.
4. Ease of portability to different operating systems to future-proof the system.

Considering those 4 criteria, Table 1 presents the best candidates for serving as programming languages of the middleware, ranked by popularity according to the TIOBE index [3]:

Programming language	**Pros/Cons**
C++	Supported on all OS platforms, produces the highest performing object code (also on low power and embedded CPUs) and low level access to sensors, for example to implement specific device drivers. The code can be recompiled with some efforts to run on different operating systems. (ranking is based on the popularity of both C and C++)
Java	Supported on all OS platforms, easy to program, not optimal for performance oriented SW.
C# .NET develop	Easy to program, good performance but Microsoft proprietary and Windows only
python	Supported on all OS platforms, easy to program, not optimal for performance oriented SW, allows very “loose” programming style which can result in large projects being difficult to maintain without strong discipline from all developers

Table 1: Pros and cons for the most popular programming languages

In consideration of its robust performance and the near real-time requirements for the project, the language of choice for MAMEM will be C/C++.

The chosen eye tracking system, the SMI REDn Scientific [76], offers a C API for interfacing with it using its SDK; furthermore all other popular ET Systems from SMI GmbH [69], Tobii AB [65], The Eye Tribe [71], EyeTech DS [66] support C / C++ APIs. Similarly, the chosen EEG recording system EBNeuro BEPlusLTM [75] amplifier offers a C++ SDK and this is the case for all widespread EEG recording systems.

2.3 Computing Platform Setup & Specifications

In our target system, the Eye Tracker and the EEG are sensors which can require a significant amount of processing power.

Indeed, all modern eye tracking systems are based on one or more InfraRed cameras capturing real time video streams of the user’s eyes. These video streams are usually transferred to a host PC through a high speed connection (USB 2.0 or 3.0, Firewire, GigE or CamLink) on which runs an eye tracking software. This software consists of a set of complex and usually proprietary computer vision algorithms which are executed on the host PC CPU – although a few ET devices exist which mount on-board processing capabilities.

Similarly, EEG recording systems capture the mean electrical activity of the brain in different locations of the head through a set of electrodes placed on the scalp. This activity is subsequently amplified through an amplifier that generates, what is usually referred as raw data, i.e. full bandwidth sampled signals. However, in order for these signals to become usable in a BCI system a certain amount of pre-processing steps are employed, such as denoising, artefact removal, band-pass filtering and spectral analysis. If we consider that in a BCI setting most of these operations will have to be performed on-line, and occasionally in the full range of captured signals (i.e. can range from 8 to 256), it is evident that the algorithms that are necessary to generate high-level events can take significant amount of processing resources.

On the other hand Galvanic Skin Response sensors measure single channel, slow evolving signals and do not require particularly complex algorithms to be processed; the same can be said for heart rate sensors.

With the sensor array chosen for MAMEM (one EEG, one ET and one GSR sensor) one common configuration used to minimize possible data loss would be a dual PC setup (see Figure 3), with one PC dedicated to processing the video streams coming from the ET module, and the other PC performing the signal processing pertaining to the EEG channels; the GSR sensor can be connected to either PC due to its low resource usage.

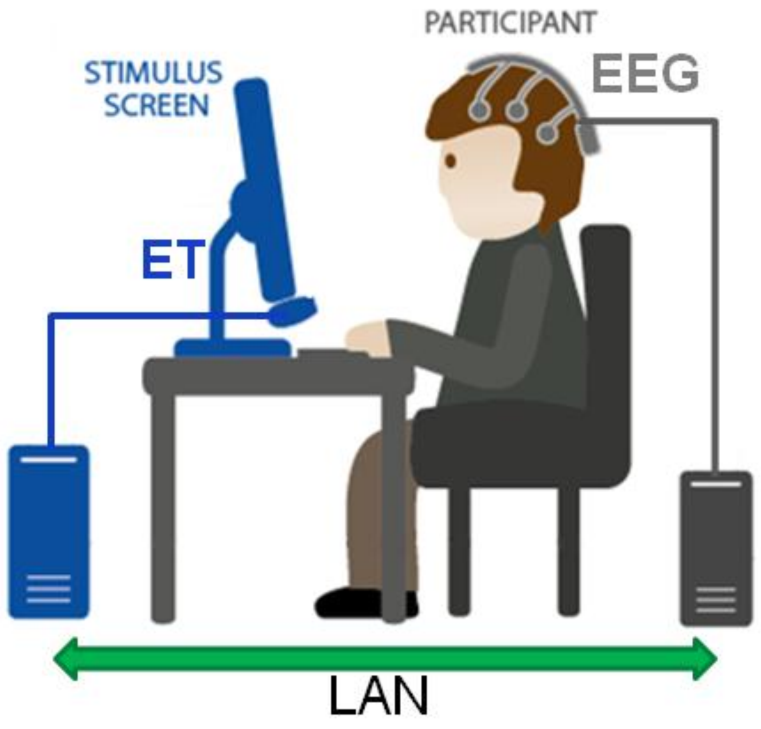


Figure 3: A “dual PC” setup, with one Eye Tracking system connected to one computer and the EEG connected to the second computer. The two computers communicate with each other via a Local Area Network.

Such a dual PC setup ensure that the concurrent load of ET and EEG processing does not (temporarily) overwhelm the system capabilities. This can happen for a number of reasons due to the non-deterministic nature of the computational load deriving from the sensors, as well as the resource scheduling (CPU, memory, etc) on commercial, non real-time Operating Systems such as Microsoft Windows or MacOS.

When such a high temporary peak in the processing load occurs, it can happen that either one of (or all) the sensor data cannot be processed within the expected time-slot, resulting in loss of data.

As it minimizes the chances of temporary data loss, such dual PC setup is a popular arrangement for scientific research which combines ET and EEG, but it also has some serious drawbacks:

- The total system cost is significantly higher when using 2 distinct PCs, and while this is acceptable in a lab environment, it would pose a higher barrier to a broad adoption of the MAMEM system as an assistive device.
- Even more important, the use of 2 PCs would limit the mobility and portability of the system, taking twice as much space, and consume twice as much energy.
- The data synchronization between the 2 systems becomes more difficult as each PC has its own clock and it is a well known fact in distributed systems that different clocks on a PC network have different offsets and tend to drift apart [4].
- The need for a (reliable) LAN connection between the PCs, ideally through a cable and router. This is due to the fact that wireless networks have unpredictable latencies which can be affected by difficult to control environmental factors such as sources of electromagnetic interferences in the same spectrum, other wireless networks active in the same space, etc.

As the goal of MAMEM is to realize an HCI interaction device, temporary data loss is not a major issue as the only practical consequence is a momentary interruption in the service offered by the device. As long as such interruption is short in duration and sporadic in occurrence it will not affect negatively the usability of the system. However, in consideration of all aforementioned drawbacks of the dual PC arrangement, it is more practical to use a single PC to process the data provided by all sensors and the respective event algorithms. The initial specs of such PC will be conservatively "high end" (i.e. heavyweight installation, see Section 2.4 to ensure a smooth functioning of all sensors without too many interruptions, and as the system matures we will propose a minimum viable configuration (i.e. lightweight installation, see Section 2.4 to lower the total system cost and possibly allow further portability.

The recommended initial specs are summed up in the following Table 2.

System Property	Initial Recommended Specs	Future Specs
CPU	Quad Core Intel Core i5 (5th generation) or better	Core i3 or Atom Z3xxx
Operating System	Windows 8.1 64bit	Windows 10
PC Form Factor	Desktop / Tower	Laptop or Tablet
Main Memory	8GB or more	
Connectivity	USB3.0 interface with Intel Chipset	
Add-ons	LPT interface add-on card to experiment with HW synchronization	

Table 2 Recommended System Specs (for the heavyweight installation, see Section 2.4

2.4 MAMEM installations

In MAMEM two types of installations are envisaged, a heavyweight and a lightweight. The goal for the heavyweight installation is to fully support MAMEM's experimental process without posing any restrictions on the hardware, stemming from the requirements of portability, cost-effectiveness and easy of use. This installation will be used to facilitate the development of the novel algorithms for translating the signals into commands and build-up the interaction SDK.

When the interaction SDK reaches a certain level of maturity, we will also implement a lightweight version of MAMEM installation. Our goal with this lightweight installation is to be: a) easily portable so as to offer MAMEM functionalities in a home environment, b) cost-effective in terms of the employed sensor devices so as to be affordable by an individual, and c) easy to setup and configure, so as to support the scenario of a non-expert using this installation for interacting with his personal computer. The main challenge for the lightweight installation is to support the functionalities of the interaction SDK while addressing the requirements of portability and cost-effectiveness.

Table 3 presents the array of sensors that have been selected by MAMEM consortium to implement the heavyweight and lightweight installation.

Installation	EEG Sensor	Eye-tracking	GSR Sensor
Heavyweight	EBNeuro BEPlusLTM [75]	SMI iViewREDN [76]	Shimmer3 GSR+ [77]
Lightweight	Emotiv EPOCH [78]	myGaze Assistive 2 [79]	Shimmer3 GSR+ [77]

Table 3: Sensor devices considered for MAMEM's heavyweight and lightweight installations

In the case of CERTH and given the availability of an existing EEG recording installation in its premises, the experimental process will be also supported by the EGI 300 Geodesic EEG System (GES 300) [80], using a 256-channel HydroCel Geodesic Sensor Net (HCGSN).

3 Middleware

As shown in Figure 4, MAMEM middleware runs on top of the sensors layer with the aim to make transparent the process of signal acquisition and de-noising, as well as its synchronization through time-stamping across different sensor devices. Moreover, through the Interaction SDK our goal is to also make transparent the process of interfacing with the function calls of the underlying operating system for accessing the navigation controls of a software application. In order to achieve this, our middleware will rely on the APIs offered by the operating systems for interfacing with their core and acquire the signals. Subsequently, the middleware will take care of synchronizing the different signals based on their timestamps and serve the synchronized signals to the interaction SDK. This SDK will incorporate the functions necessary to process the signal so as to translate it into interaction commands. Finally, the last objective that has been set for the design of our architecture is for MAMEM's middleware and SDK to facilitate their extension with new, third party interaction paradigms for touch-less interface control.

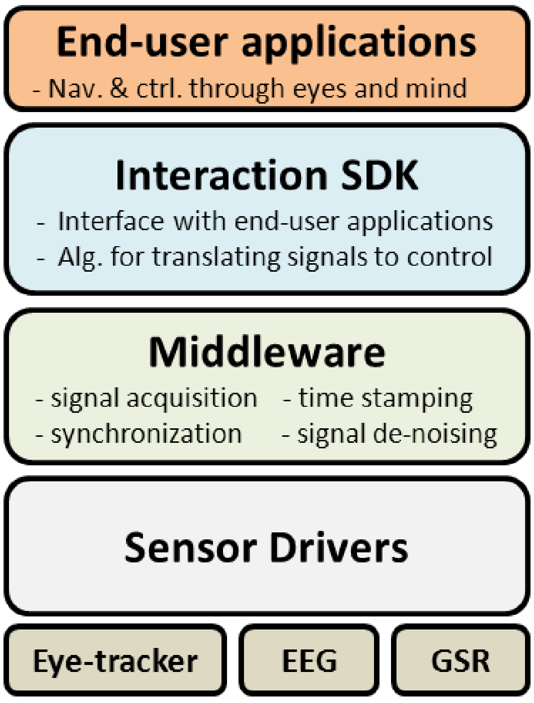


Figure 4: Role of the Middleware in the MAMEM architecture stack (figure taken from the DoA [5] – updated to reflect the organization of MAMEM's architecture into layers)

Specifically, the role of the middleware as a software component is to provide a glue layer which collects and processes the signals from the sensors, provides mechanisms to synchronize them and put them all on a single time scale, and passes the sensor data to the higher layers.

On top of the (Sensor) Middleware sits another abstraction layer, called Interaction SDK, which receives the sensor data, passes it to the specialized data processing algorithms which generate events out of the raw sensor data, and passes the events with their own timestamps to the application layer.

In this sense, the Interaction SDK takes also the role of mediating between the middleware and the end-user applications, so we can speak of a Generalized Middleware which includes both the sensor middleware and the Interaction SDK (Figure 5).

The scope of this section is focused on the (sensor) Middleware, which will provide the foundation upon which the higher interaction layers and applications will be built. Sections 4 and 5 provide a discussion for the interaction and application layers.

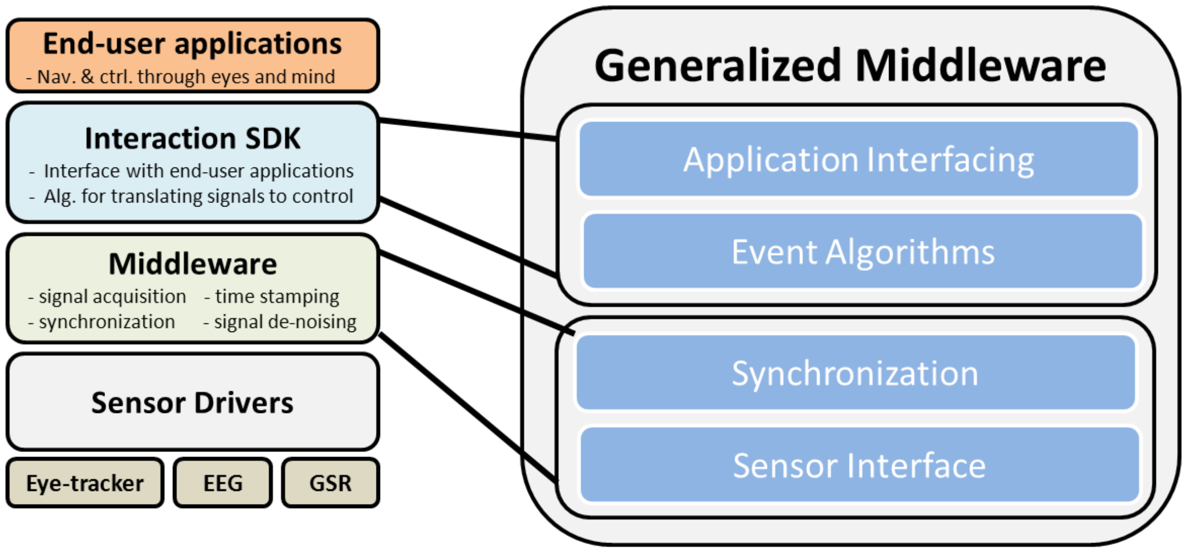


Figure 5: Generalized Middleware Architecture

3.1 Industry Standards

The natural first step to plan the middleware architecture and its realization was to search for existing solutions addressing some of the challenges posed by our project, with the intention to integrate, augment and customize for our system.

As ET and EEG interfacing and synchronization is already quite common in the research field, we found a number of ready solutions which provide the basic functionalities of data acquisition, synchronization, time stamping, and data transport.

Significantly different techniques exist, stemming from different requirements, especially in the field of synchronization; a summary of this will be provided in Section 3.5 . Table 4 discusses the pros and cons for some of the existing solutions.

Solution	Discussion
Brain Vision Analyzer [6]	Commercially available software package from Brain Products GmbH [6]: it performs automatic synchronization of EEG and ET data on the basis of common TTL event markers (See Chapter 3.5.1); however the synchronization is realized offline at the end of the experiment and hence is not suitable for our project.
EYE-EEG [1]	A similar solution, but open source, is **EYE-EEG** [7], a plug-in for EEGLAB [8] which is a Matlab toolbox for processing and analyzing EEG

	data. Synchronization is based on HW TTL markers and is offline; a number of manual steps are required to format the data in a way that it can be imported and processed then in EEGLAB.
Acqknowledge [9]	From BIOPAC [9] is another widely adopted commercial software for analysis, recording and synchronization of EEG signals; it offers a large number of predefined filters and analysis routines, has built-in support for ECG signals, however it also works only offline.
VRPN [11]	Middleware designed to offer a network-transparent interface between a set of different sensors possibly located on multiple host PCs. It is real-time, multi-platform (Windows, OSX, Linux, Android) and it is generally possible to support new devices by writing appropriate plugins, however its main field of application, as the name suggests, is Virtual Reality systems, and has a broad support of drivers and plug-ins only for motion trackers (e.g. Microsoft Kinect, 3D Mice, OptiTrack, etc).
Labstreaminglayer [12]	Open source middleware specifically developed with the goal to provide a unified collection of measurements from heterogeneous sensors to be used in research applications. It is multi-platform (Windows, Linux, OSX), supports wrapper interfaces for the most common programming languages (C, C++, Python, Java, C#, Matlab), can work online as well as on recorded data, can seamlessly transport data across a network of PCs, provides means of timestamping and synchronization of sensor data, and already supports a large number of EEG, ET, mice and motion capture sensors.

Table 4: Existing solutions for serving as the core of MAMEM's middleware

Since LabStreamingLayer already offers a significant portion of the required functionality required for MAMEM middleware, it has been decided to become a core component of our architecture and be used to collect data from the sensors, data transport and synchronization.

3.2 Lab Streaming Layer

Lab Streaming Layer (LSL) is a system for the unified collection of measurement time series in research experiments that handles both the networking, time-synchronization, (near-) real-time access, as well as (optionally) the centralized collection, viewing and disk recording of the data. The **LSL distribution** consists of two main components:

a) The core transport library (liblsl) and its language interfaces (C, C++, Python, Java, C#, MATLAB), as shown in Figure 6:. This library that constitutes the heart of LSL is general-purpose and cross-platform (Win/Linux/MacOS, 32/64), satisfying the platform independent requirement that we have set for MAMEM's middleware.

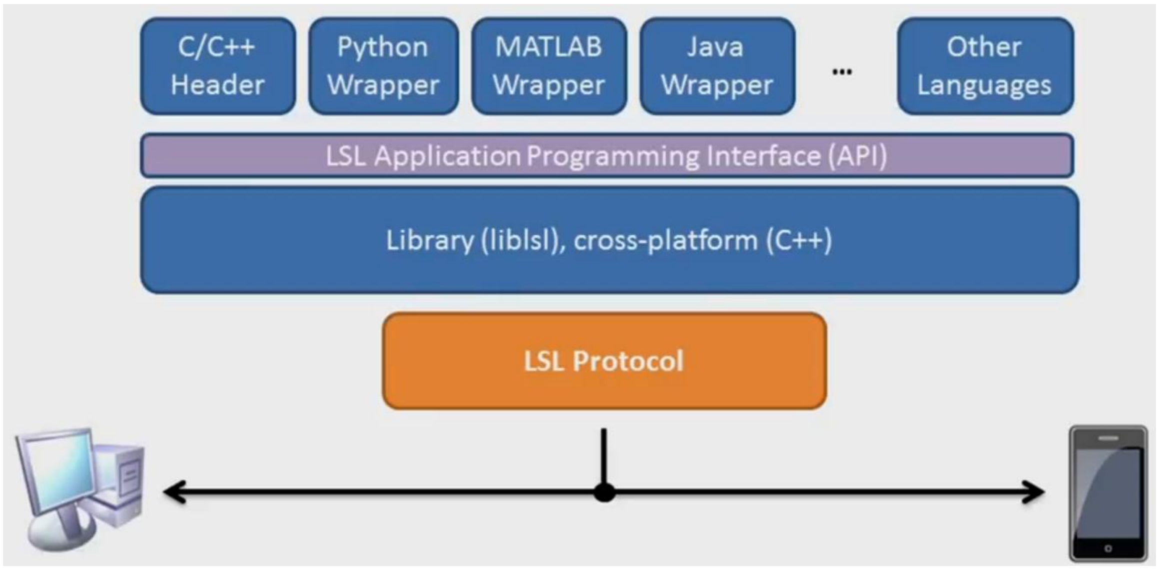


Figure 6: Core transport library of LSL (source: [34])

b) A suite of tools built on top of the library, including a **recording program**, **online viewers**, **importers**, and apps that make data from a range of **acquisition hardware** available on the lab network (for example audio, EEG, ET, or motion capture), as shown in Figure 7. This suite of tools makes evident the simplicity of extending LSL with new drivers (e.g. for supporting new sensor devices), or communicating with third party tools (e.g. like SDKs for processing, or viewing the signals) and satisfies the requirement of adopting a modular and extendable architecture.

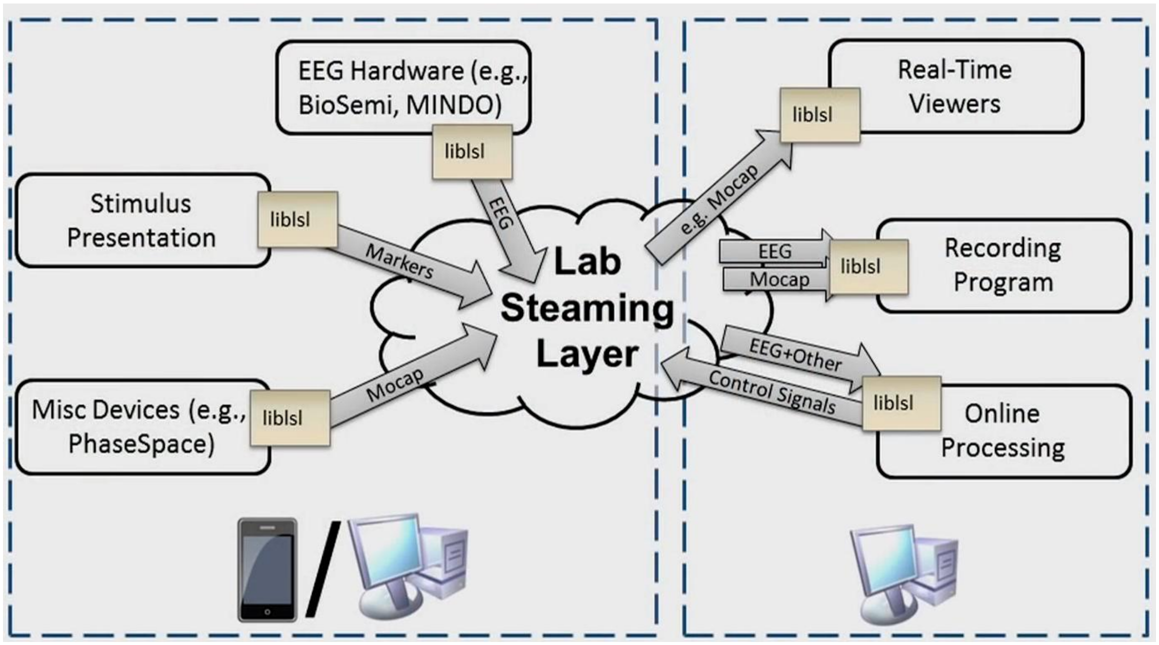


Figure 7: Network view of LSL (source: [34])

The lab streaming layer was originally developed to facilitate human-subject experiments that involve multi-modal data acquisition, including both brain dynamics (primarily EEG), physiology (EOG, EMG, heart rate, respiration, skin conductance, etc.), as well as behavioural data (motion capture, eye tracking, touch interaction, facial expressions, etc.) and finally environmental and program state (for example, event markers). Thus, it already supports an extended list of devices, as presented in Table 5. It is evident from this Table that LSL covers a wide range of existing hardware devices and, to a large extent, covers the hardware-agnostic requirement that has been set for MAMEM's middleware.

EEG Hardware (un-tested systems are marked with u)
ABM B-Alert X4/X10/X24 wireless (u)
BioSemi Active II Mk1 and Mk2
Brain Products ActiChamp series
Brain Products BrainAmp series
BrainVision RDA client
Cognionics dry/wireless
Enobio dry/wireless (u)
g.Tec g.USBamp
g.Tec g.HIamp (u)
MINDO dry/wireless
Neuroscan Synamp II and Synamp Wireless (u)
EGI AmpServer
BEPlusLTM (as part of MAMEM)
Emotiv EPOCH (as part of MAMEM)
Eye Tracking Hardware (untested systems marked with a (u)
SMI iViewX
SMI Eye Tracking Glasses
Tobii Eye trackers (u)
SR Research Eyelink (very basic)
Custom 2-camera eye trackers (with some hacking)
Human Interface Hardware
Computer mice, trackballs, presenters, etc.
Computer keyboards
DirectX-compatible joysticks, wheels, gamepads and other controllers
Nintendo Wiimote and official expansions
Motion Capture Hardware
PhaseSpace
NaturalPoint OptiTrack (some versions)
Microsoft Kinect
AMTI force plates with serial I/O
Multimedia Hardware
PhaseSpace

NaturalPoint OptiTrack (some versions)
Microsoft Kinect
AMTI force plates with serial I/O

Table 5: Hardware supported by LSL (source: [35])

In the following we provide further details on how LSL: a) interfaces with the sensor devices, b) achieves reliable data communication, c) performs the near-time synchronization of heterogeneous signals, and d) communicates with the environment of the end-user application through event markers.

3.3 Sensor Interface

The sensor interface is the lowest layer of the middleware (see Figure 5). Its scope is to provide an abstract interface for the main sensor types used in the project – currently Eye Tracking, EEG and GSR sensors. This interface is used to connect the middleware with each sensors' SDK with the intention to acquire sensor data, provide configuration facilities, and where direct user input is required provide a user interface.

LSL defines its own generic abstract interface for sensors, based on three main concepts: a) **Stream Outlets**, b) **Resolvable functions** and c) **Stream InLets**.

Stream Outlet is a time series of data which is streamed on the "lab network" defined by LSL; the data is pushed into a Stream Outlet sample by sample in the form of chunks, can be single channel or multi-channel and formatted in common data types: integers, floats, doubles and strings. In addition stream outlets can have attached metadata in XML format, which can be used as a kind of header to describe the format and use of a certain stream.

Resolve functions: these allow to resolve streams that are present on the lab network according to content-based queries (for example, by name, content-type, or queries on the meta-data). The service discovery features do not depend on external services such as zeroconf and are meant to drastically simplify the data collection network setup

Stream Inlets: for receiving time series data from a connected Stream Outlet. Allows retrieving samples from the provider (in-order, with reliable transmission, optional type conversion and optional failure recovery). Besides the samples, the meta-data can be obtained (as XML blob or alternatively through a small built-in DOM interface).

To connect a sensor device to LSL, one has to implement a suitable wrapper (in one of the languages supported by LSL, see Section3.2) which creates a Stream Outlet object, it specifies the number and type of data channels which the sensor supports, and pushes the samples generated by the sensor through the Stream Outlet, making them available to "clients" in the LSL network.

We'll show a very simple example in C++.

- It starts by including the C++ library header:

```
#include "lsl_cpp.h"
```

- Then, we declare a stream Info object which specifies our sensor will be identified as "MyEEG", it is an EEG with 100 Hz sample rate where each sample is composed of 8 channels, each channel carrying one float (32 bit) value:

```
lsl::stream_info info("MyEEG","EEG",8,100,lsl::cf_float32,"myuid");
```

- We create a Stream Outlet object with the properties we just specified in the info object

```
lsl::stream_outlet outlet(info);
```

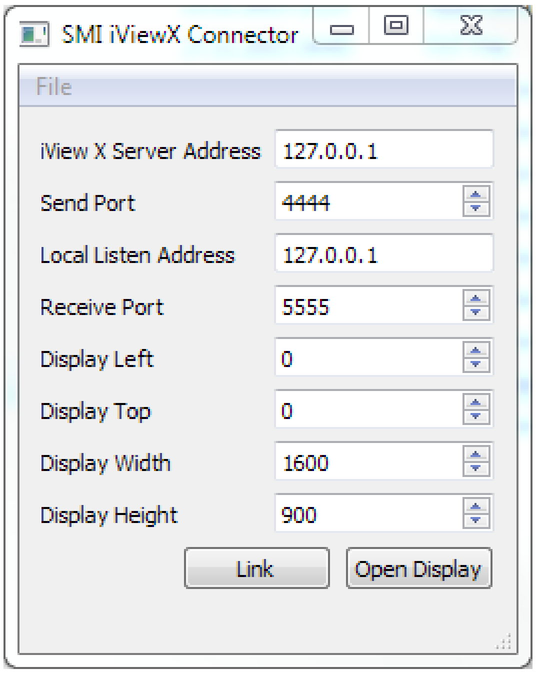
- We can now stream samples using the outlet

```
// Declare a buffer for our sample
float sample[8];

// Stream the samples into the outlet
while (true) {

    // fictious function which returns a sample from my EEG device and
    // copies it into our sample buffer
    getSampleFromMySensor(sample);

    // Push the sample into the LSL outlet
    outlet.push_sample(sample);

}
```

For more details about LSL coding guidelines, see [13].

3.3.1 Eye Tracking Wrapper Application for LSL

LSL already includes wrapper applications or "Connectors" based on the Stream Outlets concept for the major ET systems on the market, namely SensoMotoric Instruments' iViewX and iViewNG based systems [69], Tobii [65] and SR Research Eyelink [81].


SMI iViewX Connector

File

iView X Server Address	127.0.0.1
Send Port	4444
Local Listen Address	127.0.0.1
Receive Port	5555
Display Left	0
Display Top	0
Display Width	1600
Display Height	900

Figure 8: iViewX Connector Interface

As an example, the SMI iViewX Connector requires to specify the address and port at which the eye tracking server can be reached, as shown in Figure 8. Then by clicking "Link" the data will be streamed to the LSL network. A very similar type of connector is available for the Tobii systems.

It is furthermore possible to get a live visualization of the eye tracking data by configuring the display window properties (left-top corner coordinates, width and height of the display window) and by clicking "Open Display" the gaze data will be visualized as red circle over the gray display window.

The main limitation of such connectors is that they do not allow to calibrate the eye tracker, which is assumed to be already calibrated; this is not a real issue if the eye tracker is connected to the same PC where LSL is running (as in the suggested setup, see Section 2.3 , since it is possible to configure and execute the calibration by directly using the calibration SW which is part of each eye tracker SDK.

In a multi-PC environment the connector could be extended to configure and start a calibration process.

3.3.2 EEG Wrapper Application for LSL

As in the case of ET, EEG data should be sent to an outlet LSL stream and received by the middleware in an inlet LSL stream. Both the devices for the heavyweight and lightweight configuration have the functionalities to send their data trough the LSL outlet stream.

Since the heavyweight device, i.e. the BEPlusLTM amplifier by EN Neuro, was not already compliant with LSL a dedicated SDK was developed in order to easily integrate that device with LSL, as part of deliverable D2.1 [36] (see Section 5.3.1 of D2.1). At a later stage of the project, the EPOCH device will be also directly supported to connect with LSL.

EEG sensors do not need a calibration routine, as they provide low level electrical signals which need to be interpreted by complex algorithms. Such event generating algorithms are placed much higher in the SW stack. However EEG sensors require an impedance-check procedure at the beginning of a session to ensure that the electrical connection is working properly. Thus, the impedance-check procedure will be part of the EEG interface for LSL.

3.3.3 GSR and Other Sensors Interface Layer

Currently there is no supported GSR interface in the standard LSL distribution, however the mechanism to implement one using the Stream Outlets is the same as for ET and EEG sensors. It is actually easier since they do not require any configuration option.

Thus, the same approach adopted for the EEG interface layer is exploited also for the GSR module. Given that the selected device (i.e. the Shimmer3 GSR+ [77]), was not compatible with LSL, a dedicated SDK was developed to collect the bio-measurements data and send them towards LSL. See section 7.3 of deliverable D2.1 [36] for a detailed description.

3.4 Data Transport

The role of middleware in a network of heterogeneous sensors and computers is to provide a transport layer to collect the data being generated on the PCs where the sensors are attached and deliver it to the computer(s) where the user applications are running – as efficiently as possible while ensuring that no data gets lost somewhere on its way in the network.

LSL offers network transport functionality in a completely transparent manner. Stream Outlets and Inlets can be created on any PC in a local network and the user does not have to specify server addresses or network protocols. Data will be streamed from an Outlet to an Inlet in exactly the same way whether both components reside on the same machine or on two different PCs connected by a network.

LSL uses several mechanisms to ensure maximum reliability even in case of temporary network failures:

- Although UDP is used for discovery due to efficiency reasons, data samples are streamed only using TCP which is a protocol that ensures in-order, guaranteed delivery of the data
- Data is buffered both at the sender and receiver side so that a copy of it exists in case of intermittent network failures
- It provides automatic failure recovery even in case of application or computer crash

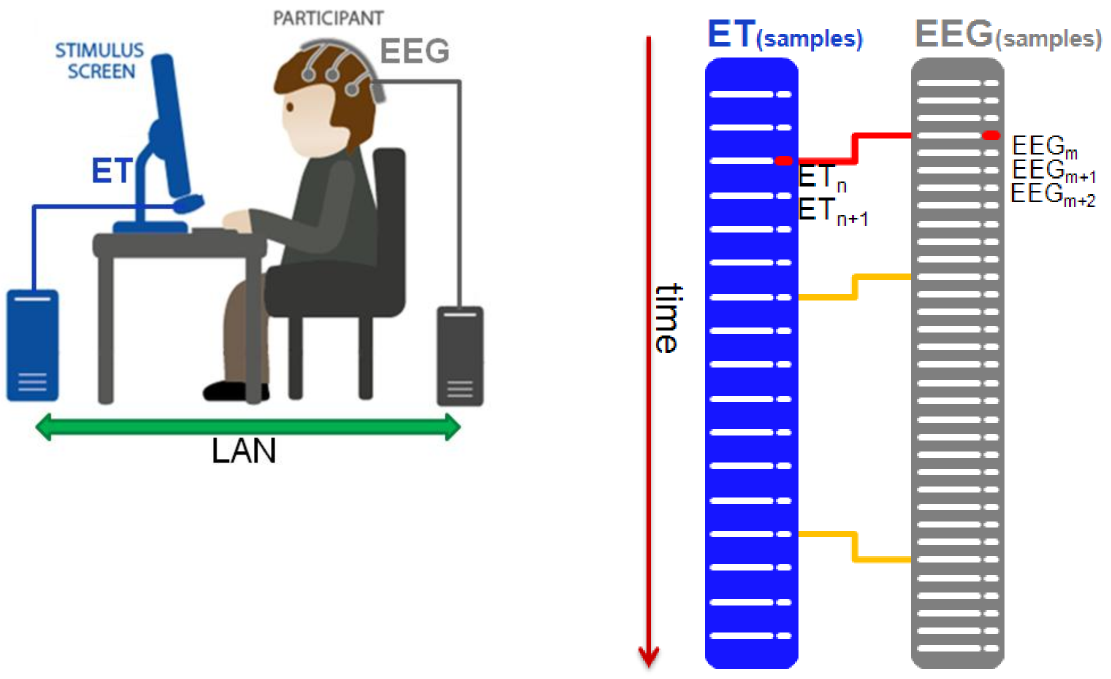
For more details see [14].

3.5 Sensor Synchronization

In a system comprising a network of heterogeneous sensors, possibly connected to multiple PCs on a local network, it is necessary to synchronize all sensor data on a common time axis in order to fuse the data to detect events. As already mentioned in Section 2.3 , in this scenario we will generally have to consider a number of factors:

- **Sensor sampling frequency**: in general will differ among the different sensors, in dependency with the characteristic frequency spectrum of the measured signal, so for example typical values can be 30Hz for the ET, 250Hz for EEG, and 1Hz for GSR.
- **Clock offset**: the time scale of each sensor in general will have a different starting point.
- **Clock drift**: even if the clock offset were synchronized, multiple clock generators in a distributed system tend to drift apart from one another as a function of temperature, clock generator technology and other environmental variables.

Figure 9 demonstrates how two signals can be de-synchronized in a dual-PC setup.



(a) (b)

Figure 9: (a) System setup comprising an ET and an EEG sensor. (b) Corresponding representation of the sensor data streams – ET (blue), EEG (gray) with different sampling frequencies and offsets. If the timeline were synchronized, sample ET_n would correspond to sample EEG_m.

3.5.1 Hardware Synchronization

Conceptually, the easiest and most precise way to synchronize a set of sensors is to use HW synchronization. Ideally this can be done by using a single clock generator to simultaneously trigger the signal acquisition of all sensors. This approach is however often not feasible, because it would require all sensors to be already designed to accept a certain external clock signal, or to specifically design a set of sensors for this specific purpose. Next to this, there are also some practical limitations:

- Camera based systems using CMOS sensors such for example ET cannot run at their maximum frame rate when the image acquisition is triggered by an external signal, because this prevents the overlapping of image acquisition between frames, which happens when the cameras have their own clock and work in Free Running Mode (see Figure 10).

- Even with a common clock source, if the transmitting lines are long enough and have different lengths, clocks will drift apart at the destination. This drift will be a function of line length, material, temperature and other factors influencing the travelling velocity of electrons in the medium.
- Currently no eye tracking system on the market allows the image capture to be directly triggered by an external signal.

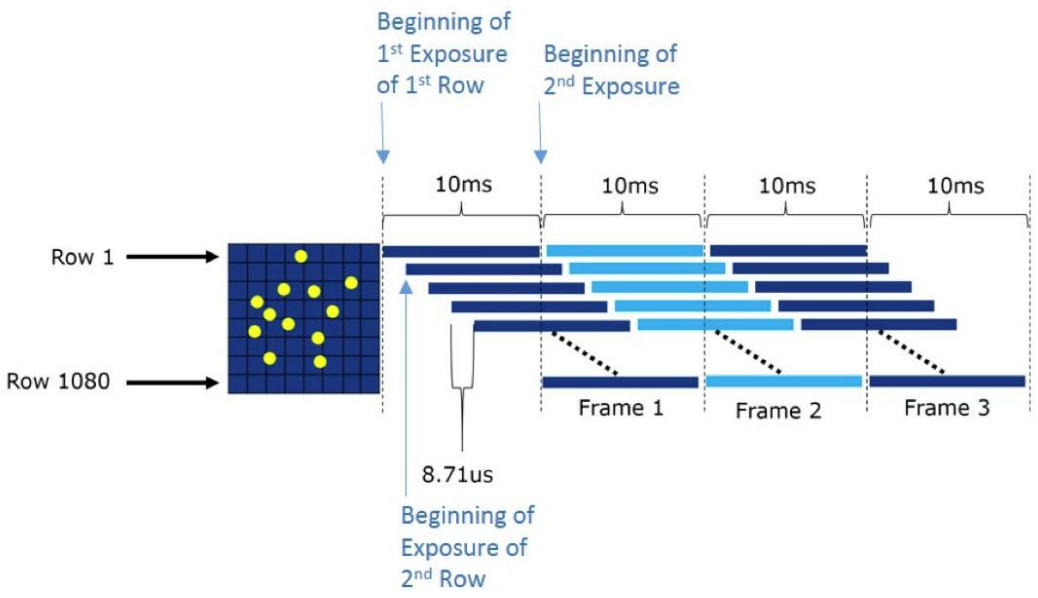

Beginning of 1st Exposure of 1st Row
Beginning of 2nd Exposure
10ms
10ms
10ms
10ms
Row 1
Row 1080
Frame 1
Frame 2
Frame 3
8.71us
Beginning of Exposure of 2nd Row

Figure 10: "Free running mode" image acquisition in a CMOS sensor with rolling shutter (source [15]). Since the time of the acquisition of contiguous frames in a sequence can be partially overlapped, the highest achievable frame-rate is higher than in triggered image acquisition.

3.5.2 Hardware Synchronization with External Trigger Events

A popular way to achieve HW based synchronization between EEG and ET systems is to use external trigger event markers.

This is achieved by having one device send the other(s) a series of "event markers", typically using a TTL logic (Transistor-Transistor Logic) electrical signal [16] (see Figure 11 and Figure 12) using an LPT (parallel port) interface. One example of this could be a marker signalling the occurrence of an event such as the beginning of an experiment, or the display on a monitor of a certain stimulus. The receiving device will then sample the signal change (from LOW to HIGH or a pulse like transition LOW-HIGH-LOW) in its own clock scale (associating it with its own timestamp) and record it in its own data stream.

By matching common events (using such markers) among the different sensor data streams, it is then possible to synchronize them, for example by shifting (clock offset), scaling (clock drift) and resampling (different sampling frequencies) one time scale with respect to the other, so that all markers appear at the same time on all data streams (see also Figure 9).

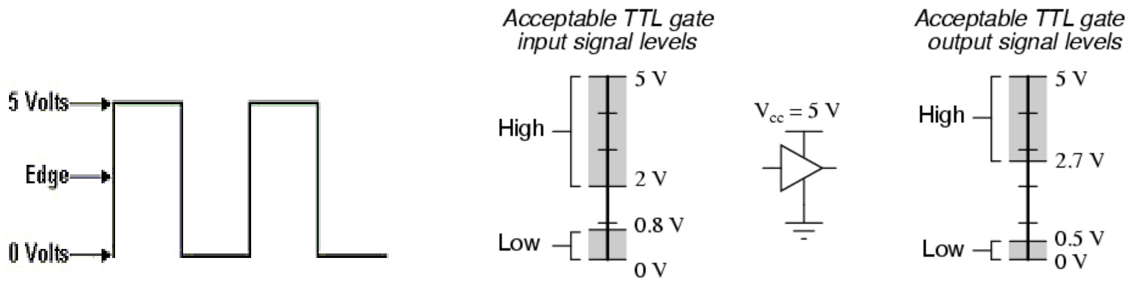

5 Volts
Edge
0 Volts
Acceptable TTL gate input signal levels
High
Low
5 V
2 V
0.8 V
0 V
V_{cc} = 5 V
Acceptable TTL gate output signal levels
High
Low
5 V
2.7 V
0.5 V
0 V

Figure 11: Example of TTL logic signal: the HIGH state is usually defined as having a voltage above 2V, while the LOW state has a voltage below 0.8V [17].

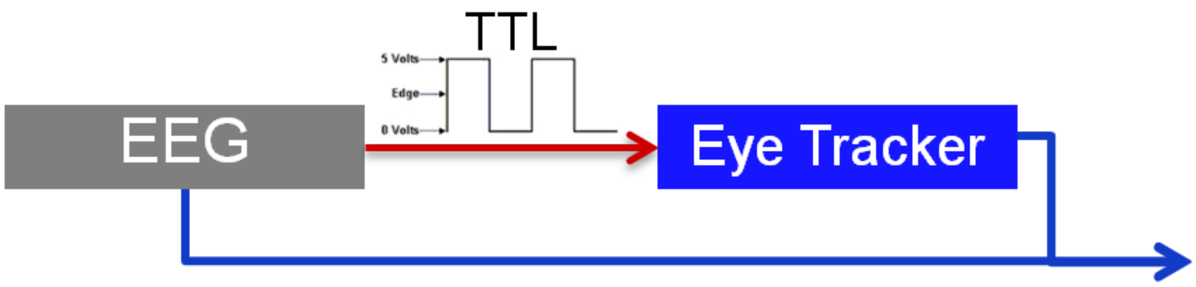


Figure 12: An EEG device sending event markers using TTL signals to an ET device

Another popular arrangement is to use a dedicated device (such as the Cedrus StimTracker [18]) to generate event triggers which are relayed to all the different sensors. These triggers can be activated by an additional type of environmental signals (Figure 13).

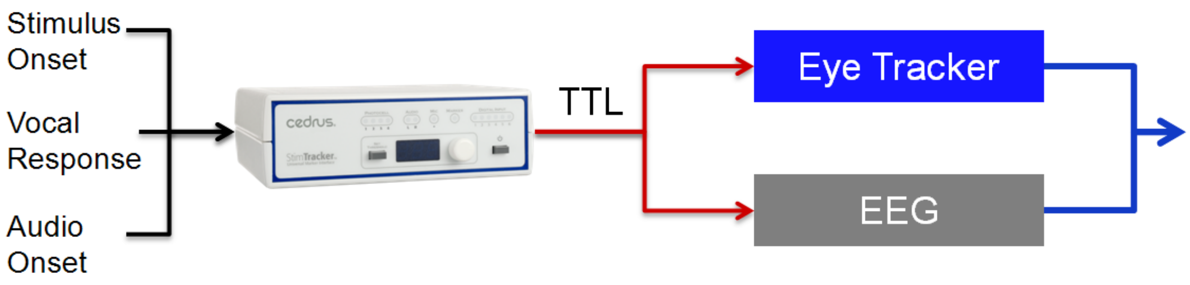


Figure 13: Example of one dedicated device generating external synchronization markers.

Such signals can be activated based on audio or light levels exceeding a certain threshold, corresponding to the onset of visual or auditory stimuli in the experiment. Another useful possibility is then for a stimulus presentation software starting an experiment or showing a particular stimulus to be able to communicate with the StimTracker using the USB interface and this generating a TTL signal marker which is then passed to the EEG and ET.

HW based synchronization using event markers can be very accurate and it is the de facto standard in research using multimodal biological sensors such as EEG and ET. However its main limitation is that by its own nature can only be performed offline at the end of an experiment (need to parse data streams to locate matching events, shifting, scaling and resampling timeline(s)) and this is not suitable for real-time human-computer interaction projects such as MAMEM.

3.5.3 Software Synchronization

The simplest form of SW synchronization is having all sensors connected to a single PC, with the PC generating a timestamp using its own central clock every time a new sample arrives. If the sensors are connected to a network of PCs however, it is generally impossible to keep all "physical clocks" of the different PCs running at the same frequency [4], so the timestamps will drift.

One easy way to deal with this problem when connected to the internet is to periodically update all the clocks using time servers broadcasting **Coordinated Universal Time** (UTC), which is defined to be the mean solar time at 0° degree longitude [2]. Using this method, it is possible to achieve a synchronization accuracy of about 1 second, provided that the synchronization with the UTC servers is happening on a frequent basis.

If higher synchronization accuracy is required, an old but still golden standard is the **Network Time Protocol** (NTP) [20]. NTP is standardized, designed to work on unreliable, variable latency networks, and it can typically synchronize all computers clocks in a network within a few milliseconds of one another, with sub-millisecond accuracy in the best case. It is not in the scope of this document to discuss the details how this protocol works, but it is based on a client computer sending messages and measuring clock offset and round trip time to a number of servers.

3.5.4 Synchronization Pitfalls

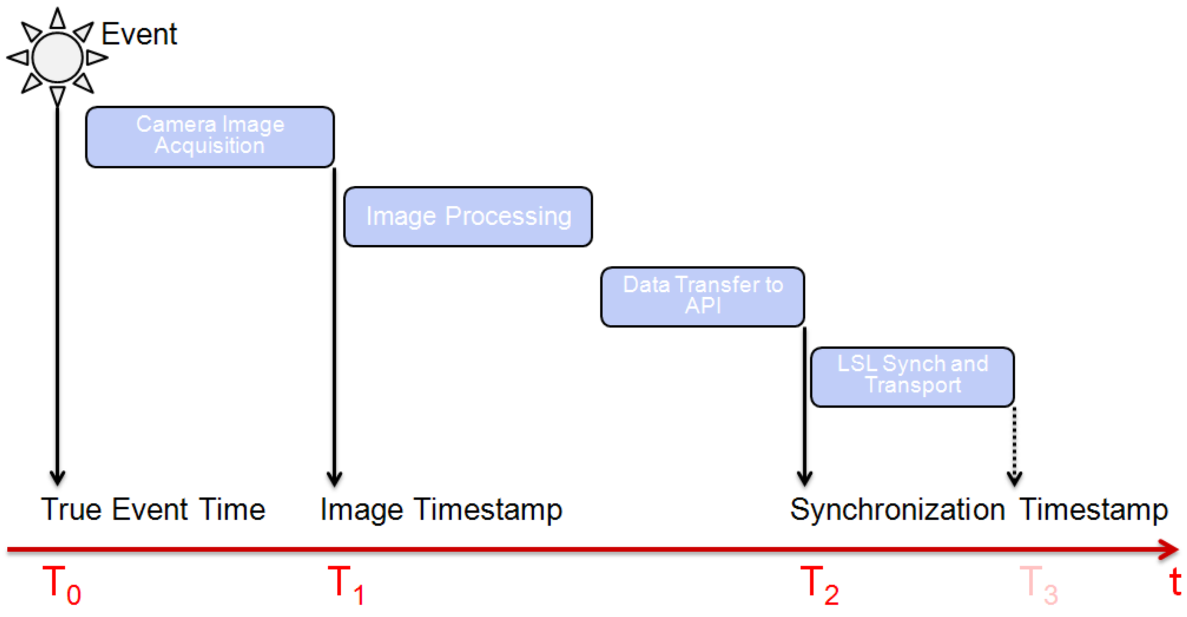


Figure 14: Internal latency of a (ET) sensor

Even having perfectly synchronized clocks, and depending on the required synchronization accuracy (for example, <1ms), there may be additional latencies to be taken into account and – when possible – compensate. In particular, sensors which are based on camera technology (such as ET) have a chain of internal latencies from the time an event occurs (e.g. an eye moves) till the time such event is reflected in the data available through the device's API.

Looking at Figure 14, there is a first latency between the time the event occurred (T_0) till the time an image has been acquired and sent to the processing CPU (T_1) where it can receive a timestamp. Such time includes the exposure time of the image sensor and the transfer time of the image from the camera to the CPU. It is usually related to the sampling frequency of the device, for example on a 250Hz ET device, it can take 4-5ms to capture and transfer an image.

After that, the image is to be processed by the CPU, this time will also be dependent on a number of factors including the speed of the CPU, the size of the image, the complexity of the processing algorithms etc. This time is typically in the order of magnitude of a few milliseconds. Finally, the data will be made available through the API (T_2) and in case of synchronization middleware such as LSL, it will receive a timestamp which will there be assumed to be the time when the event happened (T_0) but in reality will have added the internal latency (T_2 - T_o), which in many cases can be 10ms or more. Some ET systems will provide through their API for each sample the timestamp (T_1) when the corresponding image was received; this reduces the hidden latency but does not completely eliminate it.

An additional problem to consider when timestamps are being generated by a PC is timestamp **jitter**. Even if a sensor acquisition process is driven by an isochronous clock generation (which is also just an abstraction, as we know that all clock generators will drift due to environmental conditions, in particular temperature), the delivery and timestamping of samples on a PC will also be affected by a number of events of non-deterministic nature (unless a real-time Operating System is being used) dependent on temporary load of certain computer's resources such as the CPU, the bus used to transfer the data into memory, the main memory itself, etc.

The result is "noise" on the timestamps which is just being added by the PC itself and which has to be taken into consideration; if it is known however that the sample source is (approximately) isochronous and that the delta between two successive timestamps is fixed; it is then possible offline to filter the timestamps to correct or reduce jitter, if the application requires it.

Finally, camera based sensors used in real-time applications can occasionally **drop** samples. This can happen again due to temporary unavailability of PC resources, so that if an image frame is not delivered or processed on time (i.e. within the allocated time interval which is determined by the device's frame rate) then the corresponding data sample may be dropped to prevent the chain of delays in the subsequent samples.

3.5.5 Synchronization using LSL

LSL provides built-in software synchronization using a protocol which is similar to the NTP algorithm. For each sample streamed using an LSL Steam Outlet, it is possible to specify an own timestamp, if the user wants to use its own synchronization means, otherwise LSL will provide automatically a timestamp.

In case of internal sensor latencies as explained in Section 3.5.4 , if they're known it is possible to specify them in a header which is attached to the stream Outlet [21]:

```
<desc>
<synchronization>      # information about synchronization requirements
```

```
<offset_mean>           # mean offset (seconds). This value should be
subtracted from XDF timestamps before comparing streams. For local LSL
generated events, this value is defined to be zero.
<offset_rms>            # root-mean-square offset (seconds). Note that it
is very rare for offset distributions to be Gaussian.
<offset_median>         # median offset (seconds).
<offset_5_centile>      # 95% of offsets are greater than this value
(seconds)
<offset_95_centile>     # 95% of offsets are less than this value
(seconds)
<can_drop_samples>      # whether the stream can have dropped samples
(true/false). Typically true for video cameras and video displays and
false otherwise.
</synchronization>
</desc>
```

In particular, the `offset_mean` parameter is subtracted by each samples' timestamp. If the data is being recorded offline using the provided XDF format, LSL offers an XDF importer which then takes care of correcting for clock offset, missing frames (if the stream has been marked as `can_drop_samples`), resampling and jitter correction. If however the streams are being used online as in our case, it has to be considered that the timestamps of a given stream are delivered without adjustments, that is, using the clock of the computer that originally generated them.

LSL provides a very simple way to correct for the clock offset between the sender PC and the one which is receiving the data. Supposing that the receiver application has created a stream Inlet object for a certain sensor data stream (let's call this `eegInlet`), the current time offset between the sender PC's and the receiver PC's clocks, is automatically determined by LSL using an NTP-like algorithm, and is returned by simply calling the function:

```
// the time_correction() function returns the current clock offset for a
// given stream
double eegStreamClockOffset = eegInlet.time_correction();
```

It is then sufficient to add this offset to the received samples timestamps to have them synchronized with the receiver PC's clock, with one caveat. As already mentioned, clock offsets do not stay constant as the physical clocks tend to drift apart. It is then necessary to periodically call the `time_correction()` function for each stream to keep the offset up to date.

Even though the preferred setup of a single PC (see Section 2.3) minimizes the chance of having de-synchronized signals, it was deemed important for the MAMEM architecture to cover all different types of platform configuration and accommodate for network-oriented setups.

3.6 Event Generation Interface

As already mentioned in the introduction, MAMEM's software architecture consists of various different layers and modules. However, in order for these layers to operate harmonically it is important to have a joint awareness of the events that are taking place in either of the different layers. For instance, the applications layer needs to know that the EEG and ET signals have become available to the system so as to change the interface for regular mode to MAMEM mode (i.e. interface operated through eyes and mind). Similarly, in a BCI where the mental commands are facilitated by the steady-state visual evoked potentials, the Interaction SDK needs to know that the front-end application is currently in the state of presenting the different flickering boxes to the user and awaits the classification response from the system. Knowing this, the Interaction SDK could start the classification processing algorithm, which is typically in idle state to save computational resources. These are just two examples of the abstract requirement for communication across layers that can be expressed as follows:

"When a certain event takes place in one of the layers the other layers need to know about it so as to adopt their functionality accordingly"

Going back to our middleware there are two types of events that need to be handled by this layers:

a) The first has to do with the fact that that a signal stream (i.e. EEG, ET or GSR) is currently served by the middleware and can be acquired by the other layers of the architecture. In order to communicate this fact the middleware generates an event that indicates the existence of a signal stream in the local network that can be resolved by the other layers to obtain the signal stream (see Section 3.6.1
b) The second has to do with the need for the middleware to know the start and end point of an event that is taking place in another layer (usually the applications layer), so as to mark the segments of the signal streams that correspond to this event. In order to receive and transmit this kind of markers the middleware relies on the string stream event generation interface that is able to communicate strings (e.g. SSVEP-Start or SSVEP-End) at irregular sampling rate (as opposed to the signal streams where the sampling rate should be regular). See Section 3.6.2 for the description of this interface.

In the following we provide more details about the aforementioned interafaces.

3.6.1 Signal Stream Event Generation Interface

Here we are going to present how an event generation algorithm object can access the sensor data provided by the LSL network. As in Section 3.3 we introduced stream **Outlets** to send data using LSL, the dual concept on the receiving side are stream **Inlets** (see Section 3.3
.

When a stream outlet is being served in the local network through LSL (see Section 3.3 a corresponding event is generated with the intention to allow this outlet to be discovered by the other modules.

Then, an application which wants to use sensor data being streamed on the LSL network, has to first retrieve what are the currently available streams. If we want to find out a list of all streams currently active on a network, LSL provides the so-called Resolve Functions (see Section 3.3 , which return a vector of `stream_info` objects:

```
// discover all streams on the network
vector<lsl::stream_info> results = lsl::resolve_streams();
```

A stream_inlet is then created by passing a stream_info object to its constructor:

```
// Create a stream inlet associated with the first stream descriptor
// received
lsl::stream_inlet inlet(results[0]);
```

Data can then be pulled by the inlet sample by sample:

```
while (true) {
// get an 8 channel sample, each channel represented by a float
     float sample[8];
     double ts = inlet.pull_sample(sample,8);
// the returned 'ts' is the sample timestamp in the local clock
}
```

It is also possible to search (or "resolve") for streams whose descriptor matches specific properties, so for example if a stream is annotated (at the time of generating the outlet – see Section 3.3) to be an EEG stream, or using a specific name, or manufacturer, etc.:

```
// discover all EEG streams on the network
vector<lsl::stream_info> eegInfos = lsl::resolve_stream("type","EEG");
```

LSL provides an extensive documentation in its Wiki [22].

3.6.2 String Steam Event Generation Interface

Apart from the events that relate to the availability of a stream signal, LSL supports also the functionality of sending and receiving streams of strings at irregular sampling rates. As already mentioned, these strings could serve as markers indicating the start/end point of an event that is taking place in another layer (e.g. the applications layer). Below we provide the functions supported by LSL for sending and receiving a string stream.

Sending a string stream:

The following example offers a 1-channel stream which contains strings. The stream has the "Marker" content type and irregular rate. First we need declare the type of markers**:**

```
char *markertypes[] = {"Test", "Blah", "Marker", "XXX", "Testtest",
"Test-1-2-3"};
```

Then, we need to declare a new streaminfo (name: "MyEventStream", content type: "Markers", 1 channel, irregular rate:

```
info =
lsl_create_streaminfo("MyEventStream","Markers",1,LSL_IRREGULAR_RATE,cft
_string,"myuniquesourceid23443");
```

Subsequently, we need to make a new outlet (chunking: default, buffering: 360k markers):

```
outlet = lsl_create_outlet(info,0,360);
```

Finally, we can send random marker streams with the following code:

```
while(1) {
           /* wait for a random period of time */
           endtime = ((double)clock())/CLOCKS_PER_SEC +
(rand()%1000)/1000.0;
           while (((double)clock())/CLOCKS_PER_SEC < endtime);
           /* and choose the marker to send */
           mrk = markertypes[rand() %
(sizeof(markertypes)/sizeof(markertypes[0]))];
           printf("now sending: %s\n",mrk);
           /* now send it (note the &, since this function takes an array
of char*) */
           lsl_push_sample_str(outlet,&mrk);
```

```
      }
```

Receiving a string stream:

In order to receive the stream, we first need to resolve the stream of interest:

```
/* result array: info, array capacity: 1 element, type shall be EEG,
resolve at least 1 stream, wait forever if necessary */
lsl_resolve_byprop(&info,1, "type","EEG", 1, LSL_FOREVER);
```

Make an inlet to read data from the stream:

```
/* buffer max. 300 seconds of data, no preference regarding chunking,
automatic recovery enabled */
inlet = lsl_create_inlet(info, 300, LSL_NO_PREFERENCE, 1);
```

Subscribe to the stream:

```
/* automatically done by push, but a nice way of checking early on that
we can connect successfully */
lsl_open_stream(inlet,LSL_FOREVER,&errcode);
```

Display the data obtained from the stream:

```
for(t=0;t<100000000;t++) {
          /* get the next sample form the inlet (read into cursample, 8
values, wait forever if necessary) and return the timestamp if we got
something */
          timestamp =
lsl_pull_sample_f(inlet,cursample,8,LSL_FOREVER,&errcode);

          /* print the data */
          for (k=0; k<8; ++k)
                printf("\t%.2f",cursample[k]);
          printf("\n");
}
```

More details can be found at [22].

4 Interaction SDK

As mentioned in the introduction, the requirements that have been set in the Description of Actions [5] to motivate our decision on the Interaction SDK are: a) **transparent**, in terms of the communication with the middleware, b) **rich**, in terms of the supported methods and processes, c) **extendable**, in terms of adding more methods and processes, and d) **easy to access**, from the front-end applications. In the following we compare some of the most promising open source frameworks for developing BCI applications.

4.1 Industry Standards

In comparing the existing frameworks, our intention has been to evaluate the appropriateness of each solution based on the aforementioned criteria. Table 6 summarizes the results of this comparison, as inferred from [37]:

Platform	Transparency with middleware	Richness in functionalities	Easy of use	Modularity/ Extensibility	On-line processing
BioSig [38]	No real-time hardware or computation support	Large amount of functionalities for statistic and time-series analysis	Not very user-friendly (No GUI)	Complicate code, not very modular (MATLAB toolbox)	Offline analysis only
BCI2000 [39]	Supports a wide range of acquisition hardware (~19 systems)	Lack of advanced signal and machine learning algorithms	Fairly easy to use (solid documentation, big community)	Fairly modular (programmed in C++)	Supports real-time acquisition and analysis
OpenViBE [10]	Supports a broad range of acquisition hardware(~15 systems), communicates with LSL.	Focus on basic signal processing building blocks (weaker support for complex information flows)	Very user-friendly design (allows visual programming and dataflow programming)	Implemented in modular C++ but relatively hard to extent due to complex framework	Supports real-time acquisition and analysis
BCILAB [40]	Relatively little native support for acquisition systems (~5), though it can tie-up into middleware frameworks like LSL.	Largest collection of BCI algorithms from signals processing and machine learning	Fairly easy to use (by following the provided documentation)	MATLAB-based, complex internal framework, requires expertise to extend.	Supports real-time acquisition and analysis

Table 6: Comparing some of the most prominent open source frameworks for making BCI applications. (source: [37])

Apart from the aforementioned frameworks that have been considered as the most prominent, the full landscape of BCI frameworks should also refer to the following:

- **FieldTrip:** Popular MEG/EEG toolbox for online features
- **xBCI:** New C++ framework focused on online operation, GUI-centric, cross platform.
- **BF++:** Mature BCI framework providing offline analysis and modelling with UML and XML.

- **TOBI:** Protocol suite for BCI interoperability and data acquisition
- **PyFF:** Python-based BCI stimulus presentation system.
- **BBCI:** In-house, very comprehensive MATLAB-based system
- **BCI++:** Relatively new C++ system, focused on human-computer interaction and virtual reality (still in a rather immature state).

In deciding about the most appropriate framework to base the Interaction SDK of MAMEM we have only considered the frameworks presented in Table 6, since they are considered as the most mature and well-maintained. Despite its extensive set of methods for statistics and time-series analysis, **BioSig** was considered inappropriate for our purposes dues to its lack for making transparent the communication with the underlying middleware (or hardware), as well as its lack of support for real-time analysis. On the other hand, our decision to not follow the option of **BCI2000** was primarily driven by the lack of support for advanced signal processing algorithms, as well as it's moderate performance on user-friendliness. Finally, in selecting between **BCILAB** and OpenViBE, both of them are able to communicate with our middleware (i.e. LSL), as well as to support real-time acquisition and analysis. In addition, both of them are rather hard to extent, while BCILAB appears to be much richer in terms of the already implemented functionalities. However, our decision to favour **OpenViBE** for becoming the basis of MAMEM interaction SDK was its ability to support visual and dataflow programming in a very user-friendly fashion, which makes is accessible even to non-experts (e.g. clinicians). Recalling that one of MAMEM's main objectives has been to offer a framework for allowing developers and interface designers to build their own multi-modal interaction applications, the philosophy behind OpenViBE was considered to more consistently reflect the rationale of MAMEM. Finally, we should also mention that OpenViBE is written in C++ (which is in accordance of what has been considered as best practice in Section 2.2 runs on both Windows and Linux and it can smoothly communicate with LSL.

4.2 OpenViBE

OpenViBE [41] has been implemented with the purpose of designing, testing and using brain-computer interfaces. It supports real-time processing of brain signal and it can be used to acquire, filter, process, classify and visualize brain signals in real time. OpenViBE is free and open source and works on Windows and Linux operating systems.

The main application fields of OpenViBE are medical (assistance to disabled people, real-time biofeedback, neurofeedback, real-time diagnosis), multimedia (virtual reality, video games), robotics and all other application fields related to brain-computer interfaces and real-time neurosciences. The most interesting characteristic of OpenViBE is that it can be used either by programmers, or from people not familiar with programming, such as medical doctors, or clinicians. This feature makes OpenViBE particularly attractive for developing BCI applications.

In the following, we provide more details about OpenViBE in terms of the offered processes and features, the list of supported hardware devices, as well as the interfaces that are used to communicate with sensor devices or any other software that can serve as middleware (i.e. LSL).

4.3 Visual programming, processing and visualization module

4.3.1 Visual programming

In the following we provide a short description of the basic concepts behind OpenViBE as provided in [42]. The purpose of OpenViBE is to get data from the acquisition device through the *Acquisition Server* and then send it to one or more *Acquisition Clients*. In this description we will consider the *Acquisition Client* to be the OpenViBE Designer (i.e. the graphical user interface offered by the framework to support visual and dataflow programming). The Acquisition Server and the clients (Designers) can be either on the same machine or different machines on the same network, or any combination of these. The diagram of Figure 15 explains these possibilities:

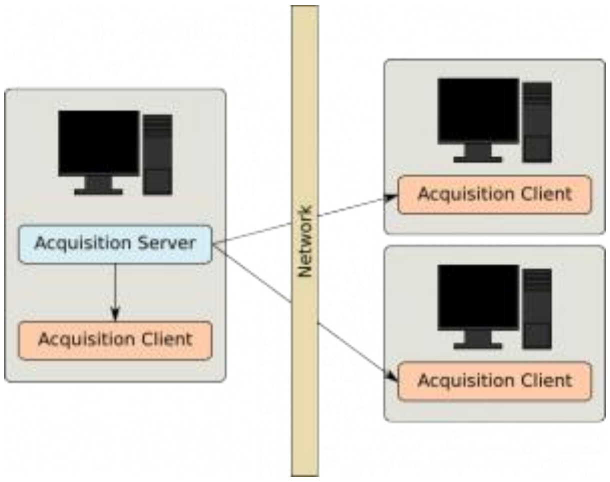
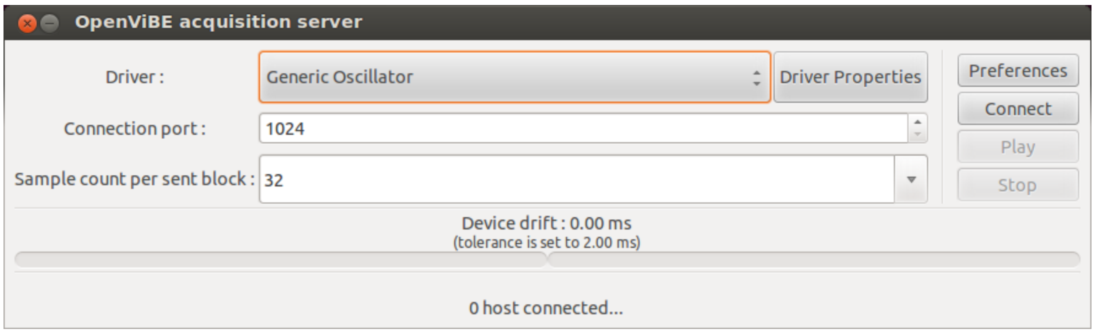


Figure 15: Diagram explaining the network topology of OpenViBE

The first step that needs to be performed in order to use OpenViBE is to run and setup the OpenViBE Acquisition Server. Upon launching this server the graphical window of Figure 16 allows you to setup the server. There are three main settings that need to be configured for setting-up the Acquisition server: a) Driver, which corresponds to the software module that takes care of the communication between OpenViBE and the sensor device, or the middleware (see Section 4.4), b) Connection port, which specifies the port that will be used by the Acquisition Server to stream the generated signal, and c) Sample count per sent block, which determines the number of samples composing a chunk received from the stream. Further configuration options can be provided through the “Driver Properties” and “Preferences”.



Driver : Generic Oscillator
Connection port : 1024
Sample count per sent block : 32
Device drift : 0.00 ms
(tolerance is set to 2.00 ms)
0 host connected...

Figure 16: OpenViBE Acquisition Server Configuration Window

The second step is running the OpenViBE designer. Upon launching the designer the graphical window of Figure 17 appears. There are two main elements in the Designer's window: a) the scenario window on the left, and b) the box algorithm list on the right. The scenario window is used to create the signal processing chains assembled from the processing boxes that are picked from the list on the right.

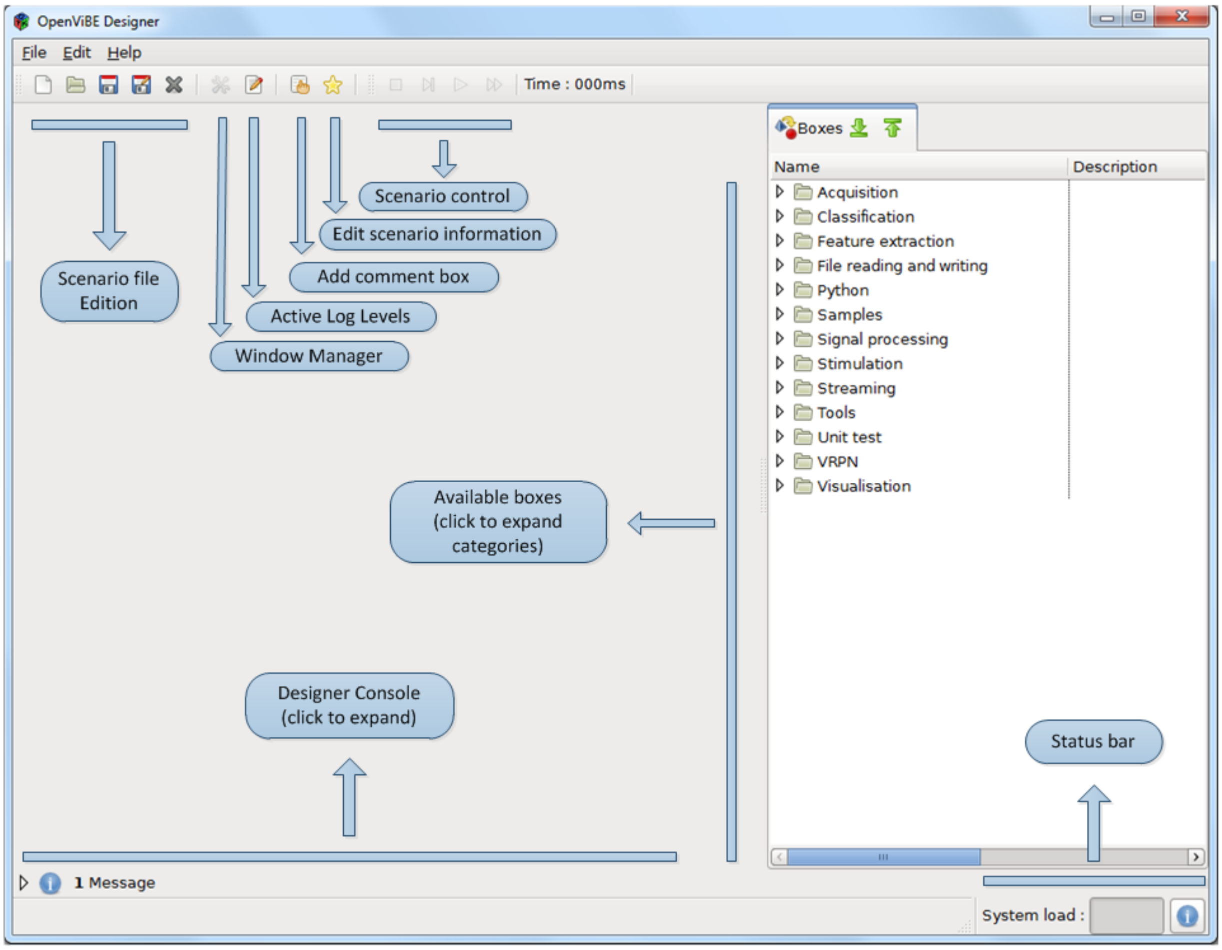
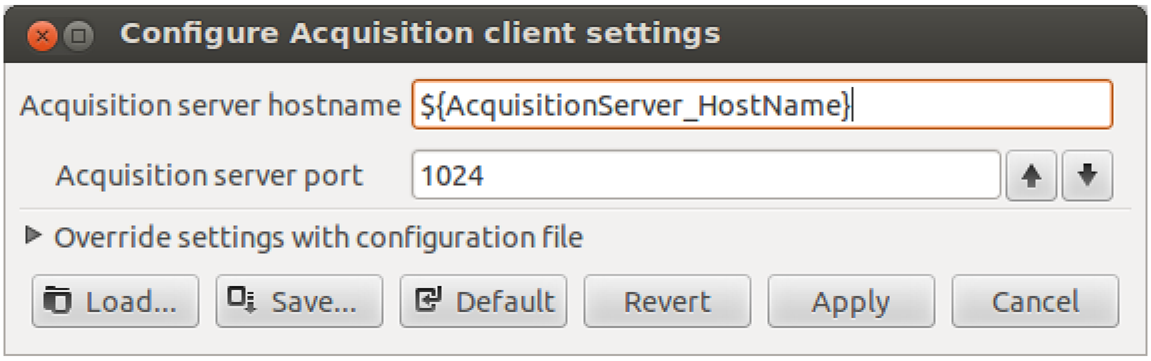


Figure 17: OpenViBE designer interface under Windows 7

The most fundamental of the available boxes is the Acquisition Client. The role of the Acquisition Client is to get the data from Acquisition Server and pass it on to the rest of the processing chain. There are two main configuration options that need to be set for the Acquisition Client (as shown in Figure 18): a) The hostname of the Acquisition server, and b) the port that is used by the Acquisition Sever to stream the generated signal.



Configure Acquisition client settings

Acquisition server hostname: ${AcquisitionServer_HostName}

Acquisition server port: 1024

Override settings with configuration file

Figure 18: Configuration window for the Acquisition client

By properly configuring the Acquisition Client the signal coming from the sensor device (or though the middleware) is now available for further processing. In order to do this, we can pick one of the processing boxes that lie on the right panel of the OpenViBE Designer. There is already a rather complete list of methods (see Section 4.3.2) that can be used for various purposes. The most trivial of these methods is to simply display the signal. In order to do this, the Signal Display box should be dragged into the scenario. This box is able to display the signal it gets on the input. Thus, if we connect the Acquisition Client box with the signal display box, as show in Figure 19, we have an example scenario that takes the signal from the sensor device and displays its waveform in the OpenViBE designer. In order to do this, we need to go back in the Acquisition Server and click on the Connect and Play buttons, so as to start sending the data over the network and also press the Play button in the designer in order to get the signal displayed.

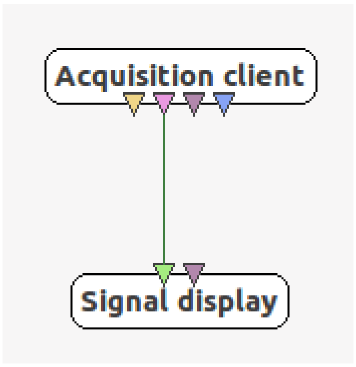


Figure 19: Minimal scenario of having a signal acquired by the device and displayed by the designer.

4.3.2 Processing boxes

As we have seen in the OpenViBE Designer, the processing scenarios are made of boxes that are dragged and dropped in the scenario panel. These boxes are used to implement all the different functionalities offered by the framework, ranging from signal acquisition and network Input/Output operations, all the way to signal processing, classification and visualization. Indeed, Table 7 presents the full list of processing boxes as provided in [43].

Acquisition and network IO	Acquisition client LSL Export LSL Export (Gipsa) OSC Controller TCP Writer
Classification	Classifier processor Classifier trainer Voting Classifier
Data generation	Channel units generator Noise generator Sinus oscillator Time signal
Evaluation	General statistics generator

	Classification • Classifier Accuracy Measure • Confusion Matrix • Kappa coefficient • ROC curve
Feature extraction	Feature aggregator
File reading and writing	SharedMemoryWriter Signal Concatenation BCI2000 Brainamp CSV EDF GDF OpenViBE • Electrode localisation file reader • Generic stream reader • Generic stream writer
Scripting	Lua Stimulator Python scripting
Signal processing	Averaging: • Epoch average • Signal average Basic: • AutoRegressive Coefficients • Channel Rename • Channel Selector • Crop • Downsampling • Epoch variance • Hilbert Transform • Identity • Matrix Transpose • Min/Max detection • Quadratic Form • Reference Channel • Signal Decimation • Signal Differential/Integral • Simple DSP • Stream Synchronization Connectivity: • Connectivity Measure Denoising: • EOG Denoising • EOG Denoising Calibration Epoching :

	• Stimulation based epoching • Time based epoching Filtering: • CSP Spatial Filter Trainer • Common Average Reference • Modifiable Temporal filter • Regularized CSP Trainer • Spatial Filter • Temporal filter • xDAWN Spatial Filter Trainer Independent component analysis: • Independent component analysis (FastICA) Spectral analysis: • Frequency Band Selector • IFFT • Spectral analysis (FFT) • Spectrum Average Statistics : • Univariate Statistics Wavelets: • Discrete Wavelet Transform • Inverse DWT Windowing: • Windowing functions
Stimulation	• Clock stimulator • Keyboard stimulator • P300 Identifier Stimulator • P300 Speller Stimulator • Player Controller • Run Command • Sign Change Detector • Sound Player • Timeout
Streaming	• Signal Merger • Stimulation Voter • Stimulation multiplexer • Stream Switch • Streamed matrix multiplexer
Visualisation	Basic: • Level measure • Matrix Display • Power spectrum display • Signal display • Time-frequency map display Presentation: • Display cue image

	• ERP plot • Graz visualization • P300 Identifier Card Visualisation • P300 Magic Card Visualisation • P300 Speller Visualisation Topography: • 2D topographic map • 3D topographic map Volume: • Voxel display

Table 7: List of processing methods available in OpenViBE (source: [43])

It is evident from Table 7 that OpenViBE covers the full spectrum of methods that are necessary to implement a BCI application. Apart from the typical modules of signal processing, feature extraction and classification, it also offers functionalities related to Stimulation and Visualization. Based on the above, our intention with MAMEM Interaction SDK is to make full use of what has been already implemented in OpenViBE and extent when necessary with MAMEM-specific functionalities.

4.3.3 Extending OpenViBE with new Boxes

As already mentioned, our goal within MAMEM is to rely on what has been already implemented in OpenViBE and extent when necessary with MAMEM-specific processes and methods. The creators of OpenViBE have made available extensive documentation on how to extent their framework with new boxes [44], how to ensure the communication between the different boxes through messages [45] and how to make these boxes available in the OpenViBE Designer [46]. Our intention is to rely on this documentation for incorporating the MAMEM-specific functionalities into the OpenViBE framework.

4.4 Supported acquisition devices

Based on its creators, OpenViBE supports over 30 acquisition devices (report up to Oct. 2015) [47]. By using the interface of the Acquisition Server we can switch between any of the supported EEG devices without the need to do any modification in the processing chain. Table 8 presents the full list of supported hardware devices, as provided by the creators of the platform in [47].

Manufacturer	**Amplifier**	**Driver Name**	**OS**
ANT	Neuro ASALAB EEG / ERP amplifier	Either MindMedia Nexus32B or TMSi drivers	⊞
	Other TMSi derived devices	Either MindMedia Nexus32B or TMSi drivers	⊞
ANT/EEmagine	EEGO	EEGO	⊞
Biosemi	Active Two MkI & MkII	Biosemi Active Two	⊞

Manufacturer	Amplifier	Driver Name	OS
BrainMaster	Atlantis	Brainmaster Atlantis and Discovery	🪟
	Discovery	Brainmaster Atlantis and Discovery	🪟
Brain Products	V-Amp	Brain Products V-Amp	🪟
	actiCHamp	Brain Products actiCHamp	🪟
	QuickAmp	Either MindMedia Nexus32B or TMSi drivers	🪟
	BrainAmp Series	Brain Products BrainAmp Series	🪟
	All	Brain Products BrainAmp Standard (through BrainVision Recorder)	🪟 🐧
Cognionics	All?	Cognionics	🪟
CTF/VSM	MEG	CTF/VSM MEG	🪟 🐧
EGI	Net Amps 300	EGI Net Amps 300 (through AmpServer)	🪟 🐧
Emotiv	EPOC (Research Edition / raw EEG versions ONLY)	Emotiv EPOC	🪟 🐧
gTec	gUSBamp	g.Tec gUSBamp Gipsa-lab	🪟
	gMobilab+	gTec gMOBIlab+ (mutually exclusive with gUSBAmp)	🪟 🐧
LabStreamingLayer	Any LSL source with compatible streams	LabStreamingLayer (LSL)	🪟 🐧
mBrainTrain	MBT Smarting	mBrainTrain Smarting	🪟 🐧
MCS/MKS	NVX	MCSNVX	🪟
Micromed	SD LTM	Micromed SD LTM (through SystemPlus Evolution)	🪟
MindMedia	NeXus32	MindMedia Nexus32B	🪟

Manufacturer	Amplifier	Driver Name	OS
Mitsar	EEG 202	Mitsar EEG 202A	🪟
Neuroelectrics	Enobio3G	Enobio3G	🪟 🐧
Neurosky	MindSet	NeuroSky MindSet	🪟
	MindWave	NeuroSky MindSet	🪟
OpenBCI	OpenBCI board	OpenBCI	🪟 🐧
OpenEEG	MonolithEEG	OpenEEG Modular EEG P2	🪟 🐧
	ModularEEG	OpenEEG Modular EEG P2	🪟 🐧
TMSi	Any TMSI amplifiers including Refa, Porti and Mobita, with up-to-date API+bios	TMSi amplifiers	🪟
	Porti32	Either MindMedia Nexus32B or TMSi Refa32B drivers	🪟
	Refa32	Either MindMedia Nexus32B or TMSi Refa32B drivers	🪟
	Other devices	Either MindMedia Nexus32B or TMSi Refa32B drivers	🪟

Table 8: Full list of hardware devices supported by OpenViBE (source [47])

We can see that all major EEG manufactures appear in the list. However, what is more important to notice is the support provided for the LabStreamingLayer. This is essentially the most important element of OpenViBE since MAMEM architecture (see Section 7) foresees that all bio-signals will arrive to the Interaction SDK through the LabStreamingLayer. However, the fact that the chosen framework is able to directly support a wide range of EEG manufactures, automatically increases the impact of the SDK extensions that will be implemented as part of MAMEM.

4.5 Discussion

One interesting remark with respect to the eligibility of OpenViBE for serving as the Interaction SDK of MAMEM is the fact that it has been mostly designed and used for processing EEG signals (and not other types of bio-signals). Thus, the built-in support for the ET and GSR signals that are envisaged in MAMEM is very limited. Despite this fact we have still considered that OpenViBE is the best option for our Interaction SDK, due to the following reasons. The transparent communication of OpenViBE with LSL ensures that it will be straightforward for us to obtain all different types of signals (i.e. EEG, ET, GSR and Event Markers) within the OpenViBE environments in a synchronized mode. Moreover, the partners of MAMEM consortium committed in extending the framework with the processing boxes that will be necessary to implement the envisaged BCI interfaces. Thus, the choice of OpenViBE as the back-bone of MAMEM's Interaction SDK was deemed the most prominent and resource-effective option for the purposes of our project.

5 Application Layer

5.1 Generally established interaction modalities – Eye Tracking

There are different techniques in the literature for using the eye gaze for interacting with a PC, however there are no common framework to interact with eye tracking based interfaces in the best manner. Generally the interaction with an eye gaze interface can be made in a **command** or **non-command** based modes [30]. In the command based interface, gaze is used for object selection in the same way as with traditional pointing devices such as mouse. In the non-command interface the user's gaze is recorded and examined to discover the user's attention.

There are certain difficulties with using eye movements as commands in a human-computer dialogue. The behavior of the eye makes it unsuitable to replace manual input devices directly with an eye tracker. One characteristic of eye movements is that they are both controlled consciously and affected by external events. When using eyes to interact with a computer, the eyes are always "on" hence the input is continuous. However the manual input devices are active when the user desires them to be. These characteristics make it very difficult for the computer to interpret the user's intentions only from gaze focus. All gaze fixations made by the user cannot be regarded as commands. In the research community this is called the **Midas touch problem**. There should be methods for the user to confirm that a command should be executed or not. To avoid Midas touch problem it is possible to use techniques like: **Dwell time, Winks, Extra input device**. With dwell time, a selection is confirmed after the user has looked at an object a certain amount of time. When using dwell time it is important that the response time is sufficient. If the dwell time is too short, wrong selection may be made and if it is too long, the user gets frustrated. If a wink is used the interface must know if it is intentional or not. A wink with one eye is used to make it possible to separate the intentional blinks from the unintentional, which always are made simultaneous with both eyes. When the user looks at an object on the screen, the object becomes highlighted. No action is performed until the user winks with one eye – then a command is executed. Several winks can be joined to make a special command. For example a wink with left eye followed by a wink with right eye can imply "page up" in a word processor. The use of a combination of eye and hand for controlling a user interface may be the best option in most situations. This makes the use of an extra input device necessary and thus the technique is not an option in a purely gaze based interface.

An important aspect of interaction is when the screen interface should include a **cursor** that follows the user's gaze focus. If calibration, accuracy and speed of the eye tracker would be perfect, no feedback would be needed, since the user knows where he is looking. Cursor is a way of feedback; it shows the user where his gaze focus is, according to the computers interpretations. But there are also disadvantages with the use of a cursor in an eye gaze interface. If there is a flaw in the calibration, the cursor will be shown displaced from the user's real gaze focus. The user's gaze will then be drawn to the cursor, which becomes further displaced. An alternative to the use of a cursor is to highlight the items that the user is focusing on.

Another popular modality is gaze added interface when the eye tracker is used to complement the manual input devices like mouse or keyboard. In this kind of interface, the problems with gaze accuracy and Midas touch can be solved in different ways. A command can be confirmed by pressing a button. A button on the keyboard or one of the mouse-buttons, called “the gaze-button”, can be used for this purpose. Jacob et al. proposed an approach where both dwell time and a gaze-button works in parallel [27]. The user can then choose the interaction technique that he prefers. If interaction with dwell time is regarded as too slow, a command can instead be executed by pressing the gaze-button. Zhai et al. [28] considered that the human visual perception channel should not be loaded with a motor control task, like selecting or moving an object. To solve this problem they developed a method **MAGIC pointing** – Manual And Gaze Input Cascaded pointing. The technique is based on the idea that pointing and selecting should be manual tasks; while at the same time make use of the benefits with eye control. The speed of the eye makes it appropriate to use for fast movement of the pointer. In this interface the pointer is “warped” to where the user fixates his gaze. If the fixation is close to an object, fine adjustment is made with a manual input device.

Using these interaction methods, several applications have been developed in the eye gaze community, since the interest for applying eye tracking methods grows with technological progress and increment of performance and accessibility [29]. The devices are becoming sufficiently reliable and affordable to consider their use in real HCI. Many studies are focused on appropriate interaction techniques that incorporate eye movements into the HCI in a convenient and natural way. In the following we discuss some popular usability application scenarios of eye tracking based interaction.

5.1.1 Eye tracking in assistive technology

Assistive technology encourages greater independence for people with disabilities by enabling them to perform tasks that they were formerly unable to accomplish. Taking into account that most of the neuro-disabled patients can move their eyes, this can be useful for communication. In a gaze based interface the user does not need to use his hands to interact with the computer. The only input device used is an eye tracker and solely the user’s eyes control the GUI. This kind of interface is suitable for people with physical disability that prevents them from interacting with their hands. It is also a good option if the space available does not allow a keyboard or mouse to be used. Instead of using a keyboard the user can write by using an eye controlled graphical keyboard displayed on the screen.

Eye tracking can be used together with a computer to select a word from a menu. This device should be used by patient for a face to face conversation or a remote message sent via communication network. Figure 20 presents an example of such a system [24], where the keywords are selected by patient using eye tracking technique. A camera mouse can be used to move a cursor on a computer screen and to browse a menu for suggestive pictogram selection [25]. The keywords collection is organized as a tree structure having wide and short topology. The breadth first traversal method is suitable for keyword searching and for an easy and fast comeback to the upper level “Go back” images are placed at the right and left limits. An updated version of this communication system uses an eye tracking mouse (ETM) system using video glasses and a robust eye-tracking algorithm [26]. The validation of the usability and reliability of the proposed system was done by experimental procedure involving voluntaries and patients in a neurologic emergency clinic.

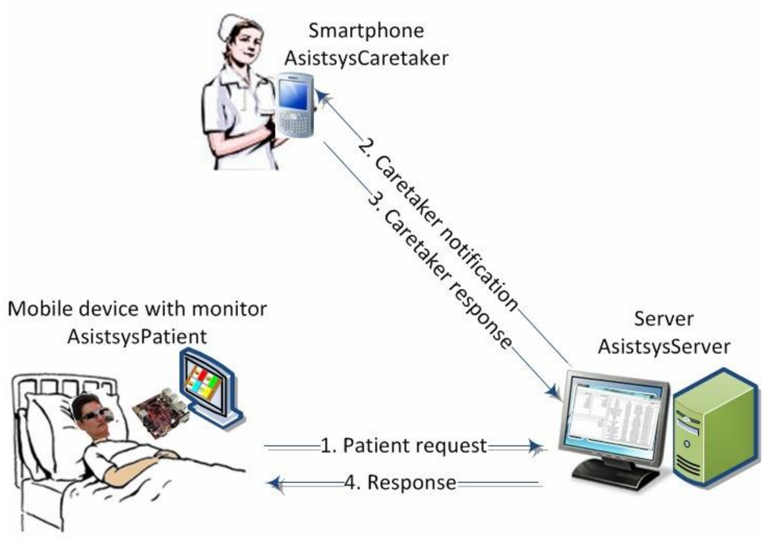


Figure 20: System communication for people with disabilities (Asistsys)

In the Eye gaze systems such as PRC Accent [68], myGaze [70] with Grid 2 software [63], Tobii Dynavox [64], a user can operate the system by looking at rectangular keys or cells that are displayed on the control screen. Through visual activation, the array of menu keys and exit keys allow the user to navigate the software independently. Through this eye tracking technology, users can operate lights and appliances remotely, control infrared devices such as televisions and stereos. For more sophisticated computer access there are other systems from, among others, SensoMotoric Instruments [69], LC Technologies [67], and Eye Tech [66]. However, following Medicare guidelines, people enrolled in hospice or living at an assisted living facility are sometimes not eligible for communication devices. For those going down the do-it-yourself route, there are new cost effective devices such as The Eye Tribe [71] which potentially enables eye control on some mobile devices. The Eye Tribe tracker is currently a development unit intended for developers only and cannot be used commercially; however it is supported unofficially by some open source software [72]. The EyeWriter Project [73] is a low cost, open source eye-tracking system allows patients to draw on a tablet using just their eyes. Vision Key [74] is one of the latest eye controlled communication that enables users to type and talk with their eyes. The system gives the users a voice by enabling them to control a speech synthesizer in the Vision Key unit or on the computer by looking at the screen. Users look at a specific word, letter or character on the chart in front of their eye and ‘type’ by holding their gaze until a selection is confirmed by a green highlight and a beep.

5.1.2 Eye tracking in e-learning

In recent years, several technologies like collaborative software, cloud computing, screencasting, virtual classroom together with different devices (e.g. mobile devices, webcams, audio/video systems) were used to facilitate e-learning development and to increase the effectiveness and accessibility of e-learning platforms. Various studies have revealed that eye tracking methods could actually improve the functionality and usability of e-learning systems: Eye Tracking Analysis in the Evaluation of E-Learning Systems, project, AdeLE project or ACM studies [23].

In the e-learning platform it is possible to capture learner behavior in real-time using eye tracking methods. The data collected via eye-tracking devices indicates the person's interest level and focus of attention. From eye position tracking and indirect measures, such as fixation numbers and duration, gaze position, and blink rate, it is possible to draw information about the user's level of attention, stress, relaxation, problem solving, successfulness in learning, tiredness or emotions. It was revealed that when using eye tracking in e-learning, the learner pays more attention to the learning system and also tends to have a higher level of motivation [23].

5.1.3 Eye tracking in gaming

Several commercial games have already explored the concept of gaze based interaction for better user satisfaction [31]. The most obvious means of eye based interaction is in pointing tasks, where the object the user is looking at is considered to be selected. For example, in the first--person shooter genre, where the field of view of the player's avatar is explicitly presented to the user. Other games build on the way humans use eye movements to manage social situations as a means for an avatar to communicate with the player. For example, In "The Legend of Zelda: The Wind Waker", the player's avatar indicates that a nearby object might be interesting by looking at it [32]. Additionally, some games require the player to explicitly control an avatar's gaze direction. For example, in Figure 21 user's gaze controls his butterfly avatar, moving over a meadow towards the horizon gathering flowers. The game is developed at University of Koblenz [82], and explores three different gaze control mechanisms. The first approach is a direct interpretation of the gaze coordinates as position for the avatar. The second one is a grid-control variation, with predefined positions for the avatar. The third approach is a mechanism that supports the players by automatically directing the avatar to a position where it will collide with the flower.

Eye movements have also been proposed as a modality for pointing within virtual environments. These systems typically correlate the user's gaze into a vector defined by virtual world coordinates. Typically, 2D gaze coordinates are retrieved from the eye tracker and then projected into the world using simple ray casting. Tanriverdi and Jacob proposed that eye movements could be used as an active pointing device for 3D object selection in virtual environments presented in a head-mounted display [33]. The eye was tracked in 2D in screen coordinates in the HMD. Ray casting was used to select the nearest object rendered to the pixel residing at the gaze coordinate, and a dwell time was used to avoid the Midas touch effect.

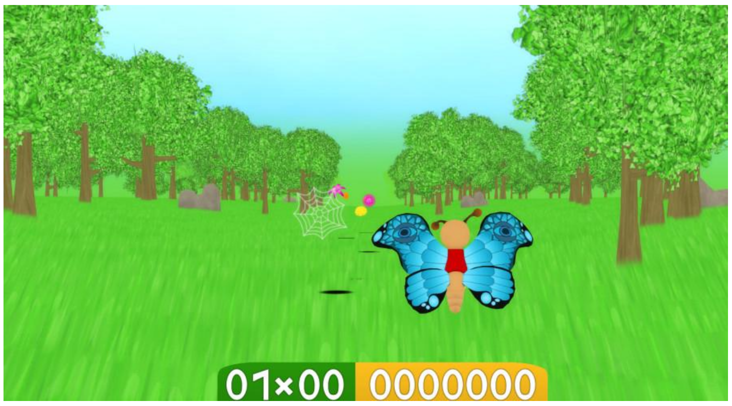


Figure 21: The avatar is controlled by the player's eyes (Schaugenau)

5.2 Generally established interaction modalities – EEG

5.2.1 BCI Systems and EEG-based interaction modalities

A general description of a BCI system is provided in Figure 22. BCI is a tool that gives us the ability to communicate with the external world without using peripheral nerves and muscles. A BCI system translates the recorded electric brain activity into output commands. To achieve that, a number of steps are performed, as indicated in Figure 22. The input to a BCI system is the electrophysiological brain activity, while the output is the device commands. The brain activity is recorded through the use of an EEG system. After that, the analysis of EEG signals is performed in order to extract the intended commands of the user.

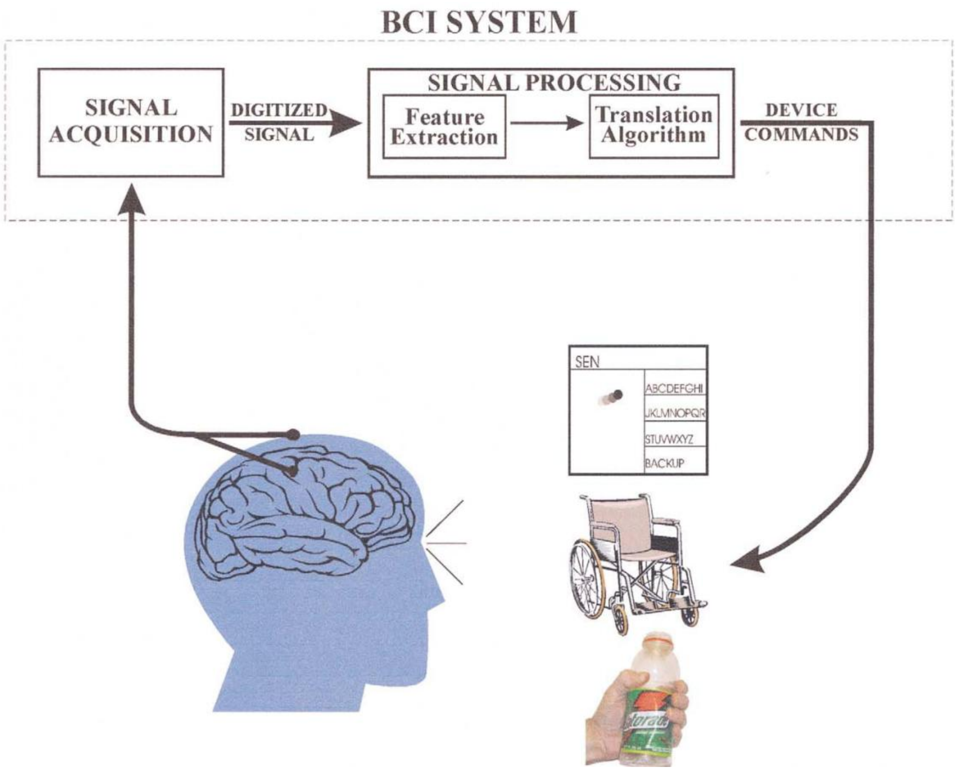


Figure 22: A general description of a BCI system (reprinted from [48])

A BCI system can be characterized in a number of ways based on the different modalities of physiological measurement (electroencephalography (EEG) [57], [58]; electrocorticography (ECoG) [59]; magneto-encephalography (MEG); magnetic resonance imaging (MRI) [60], [61]; near-infrared spectroscopy (fNIRS [62]), mental activation strategies (dependent versus independent) and the degree of invasiveness. From the above modalities, the EEG signal is the most used because of its noninvasiveness, its high time resolution, ease of acquisition, and cost effectiveness as compared to other brain activity monitoring modalities. Noninvasive electrophysiological sources for BCI control include event related synchronization/desynchronization (ERS/ERD), visual evoked potentials (VEP), steady-state visual evoked potentials (SSVEP), slow cortical potentials (SCP), P300 evoked potentials and μ and β rhythms.

5.2.2 BCI applications based on EEG

By relying on the aforementioned modalities, a number of BCI applications have been proposed in the literature based on EEG signals. For instance, one type of applications concerns the interfaces that have been developed for helping people with neuromuscular disorders to type letters. The user concentrates on a flashing letter on the computer monitor (see Figure 23). This creates an electrical change in the brain which is sent to the computer. The computer program translates the brain signal to the letter that the subject was focusing by exploiting the time information of the flashing letter and the electrical brain signal. This brain signal is called P300 (P3) wave and it is an event related potential (ERP) component elicited in the process of decision making [49]. This method has passed into market with several companies selling related hardware & software such as the "Intendix Brain-computer Interface" [50]. The same idea has also been used for painting. The user can paint by focusing on a set of buttons and control is based on the visually evoked potentials (VEP/P300) as in the previous case.

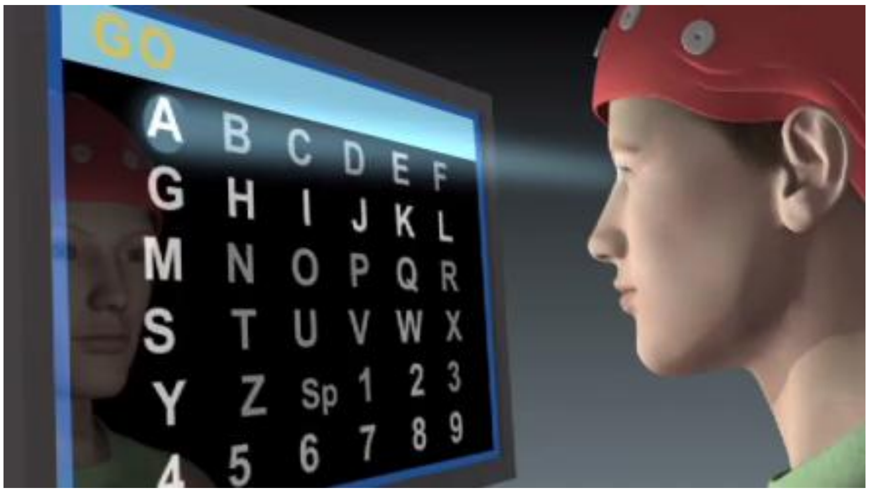


Figure 23: BCI application that allows the user to type by staring at flickering letters

Another typing middleware is the hex-o-speller that has been developed in the context of the TOBI project [51]. In this case instead of flashing symbols the software relies on a hierarchical class split and visual evoked potentials (focusing). The letters are separated in 6 classes that are shown in 6 hexagons. An arrow rotates from one hexagon to the other. When the subject focuses in one hexagon the arrow expands towards this hexagon and the selected class is split further until a single letter class is reached. Once the letter has been selected the system goes back to the first screen containing the most probable set of letters, and the user reiterates the cycle to select the following letter (see Figure 24).

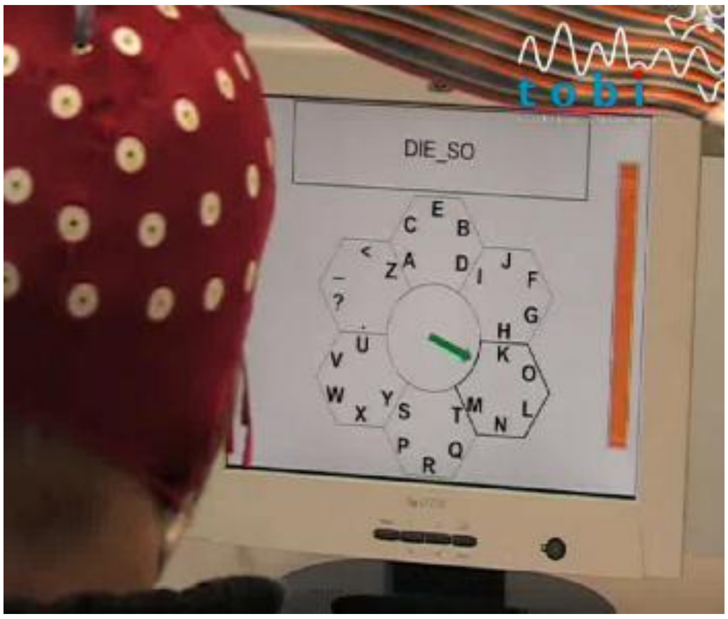


Figure 24: BCI application that allows typing based on Visual Evoked Potentials and a hierarchical class split.

Brain computer interfaces have also been used for allowing the subject to use brain signals for navigating in Virtual Reality (VR) worlds, as in the system developed by the Graz Brain Computer Interface Lab [52]. The subject imagines that he is moving one of his/her limbs, and this brain signal is translated to a command such as “right”, “left”, “button A”, etc (see Figure 25). The same lab has also designed an interface for using the “Google Earth” program. This is done by allowing the user to select the desired location using the aforementioned technique of the “moving limbs”. The same technique has been used also for audio processing by selecting the desired soundtrack and applying signal processing filters.

Figure 25: BCI application that allows navigating in the virtual world.

5.3 Native and web browser applications

What becomes evident from Sections 5.1 and 5.2 is that BCI applications are evolving rapidly and try to find their place in the market. What is also interesting is the variety of the end-user tools that are used to present the end-user applications. Indeed, the majority of the existing BCIs run as native applications (implemented in Java, C++ or some other programming language) that have been developed solely for this purpose. However, there are also BCIs that run through a web browser or through some other generalized framework. This tendency poses the requirement for MAMEM architecture to incorporate an additional *Layer* that would make transparent the communication between the back-end of our system (i.e. capturing and translating the signals) and the end-user applications. In addressing this requirement, we have incorporated in our architecture the Application-Network Layer (described in Section 6) that handles this communication through a client-server scheme relying on network sockets. This layer adds a level of transparency between the back-end of our system and the front-end applications, ensuring that MAMEM architecture will be able to support all different types of end-user applications, independently on whether they run as native applications, through a web browser or some other framework.

6 Application-Networking Layer

The purpose of the Application-Networking Layer is to provide the means of communication and data sharing between the Application Layer and the rest of the system. The communication can be realized in many different ways but most of them rely on the underlying operating system, or require the different processes to be running in the same computer (see **Table 9** for a comparison). Our intention is to have a flexible architecture and for that reason we chose to use sockets because they overcome most of the limitations described above.

Method	Advantages	Disadvantages
Communication with Files/ Named Pipes [84]	1. Doesn't rely on the OS 2. Can be established over a network	1. Slow because it utilizes hard disks. 2. Web applications cannot access the file storage of a computer directly; a web/file server must be also implemented.
Shared Memory/Memory-mapped file [85]	1. Very fast.	1. Only native applications may access the memory 2. Processes must be running on the same computer.
System Signals [86]	1. Easy to implement	1. Only specific type of messages may be passed (predefined by the operating system) 2. Only native applications are supported.
Sockets [87]	1. Do not rely on the OS 2. Communication may be established by different computers over the network 3. Well supported by many different programming languages	1. Small latency on data transfer

Table 9: Methods for communication between different software processes.

Since we are planning to support both native and web applications, we need to use a socket protocol that is supported by both platforms. There are many different protocols that can be used for native applications, but it is not the same case for web applications. Web applications are executed in a "sand box" mode inside a web browser for security reasons. This means that they do not have access in the majority of the resources related with the operating system, file storage or other APIs outside of the browser environment. The most common way for a web application to connect with external applications is to use AJAX calls, or the relatively new WebSocket protocol [53]. We chose to go with the WebSocket method since it allows bi-directional communication between a web application and a server, meaning that a web application can at the same time, send and receive messages to/from a server. This is important for our architecture since it is expected that the system listens to the String Steam of Event Markers from the Application Layer and also broadcasts the HCI Triggers back to Application Layer.

For implementing the WebSocket protocol we will utilize the Socket.IO [54] library, a library written in Javascript that supports real-time and bidirectional communication between a server and its clients. The main reason for choosing Socket.IO, even though it contradicts the best practice of using C/C++ suggested in Section 2.2 , is the widespread use and maturity featured by this library in implementing a client-server model. Indeed, the library is open-source, hosted in GitHub [83] and includes many samples that will help on speeding up the development. It is separated into a client-side and a server-side library for the node.js server. Although it has been designed for communication between web browsers and web servers, it can also be used with clients written in other languages such as C++ [55]. Finally, the library is supported by the recent versions of all modern browsers, such as Google Chrome, Internet Explorer, Firefox, Safari and Opera.

6.1 Socket.IO client-server model

In motivating our decision to consider a client-server model (as opposed to a peer-to-peer model) for implementing the socket-based communication we have considered the following. Although the Socket.IO library supports a peer-to-peer architecture [88] based on WebRTC [89] which allows direct communication between the clients, the technology is not mature enough yet and there is a high chance that we encounter problems such as browser incompatibility. In addition, we estimate that the benefit of using a peer-to-peer architecture (i.e. lower latency for passing the messages especially when streaming large amount of data) does not compensate for the increased amount of complexity and development effort. For this reason we chose to follow the more established client-server model.

The server will be based on node.js [56], an open-source and cross-platform server-side scripting runtime system. There is no limitation regarding what computer the server will be run as long as the computer is part of the local network, however the best option is to setup the server in a computer with low CPU usage from other sources. The role of the server is to act as an intermediary between the clients and more specifically to: a) Receive messages from clients, b) Interpret the messages according to the Messaging Protocol (see Section 6.2 and c) Send a response message to the appropriate client.

The clients will connect with the server when they are initialized and then will be able to freely send or receive messages from the server. In our case the client applications include a plugin for LabStreamingLayer (the Middleware), which is responsible for generating the event marker string stream, a plugin for the InteractionSDK that sends the HCI triggers to the Application Layer, and the applications themselves which mainly listen for the HCI triggers and broadcast the event markers.

6.2 Messaging protocol

The messages that will be passed to and from the server will comply with a messaging protocol common for all clients that are part of the architecture. It will be a JSON formatted string which can be parsed in order to receive the data. When a client connects to the server an identifier is assigned to it automatically. The server will preserve a list of all connected client IDs and will transmit it on demand when a client requests it. Each time a client wants to send a message to another client it will have to fetch the list of the connected clients from the server. A typical flow of communication can be seen in Figure 26.

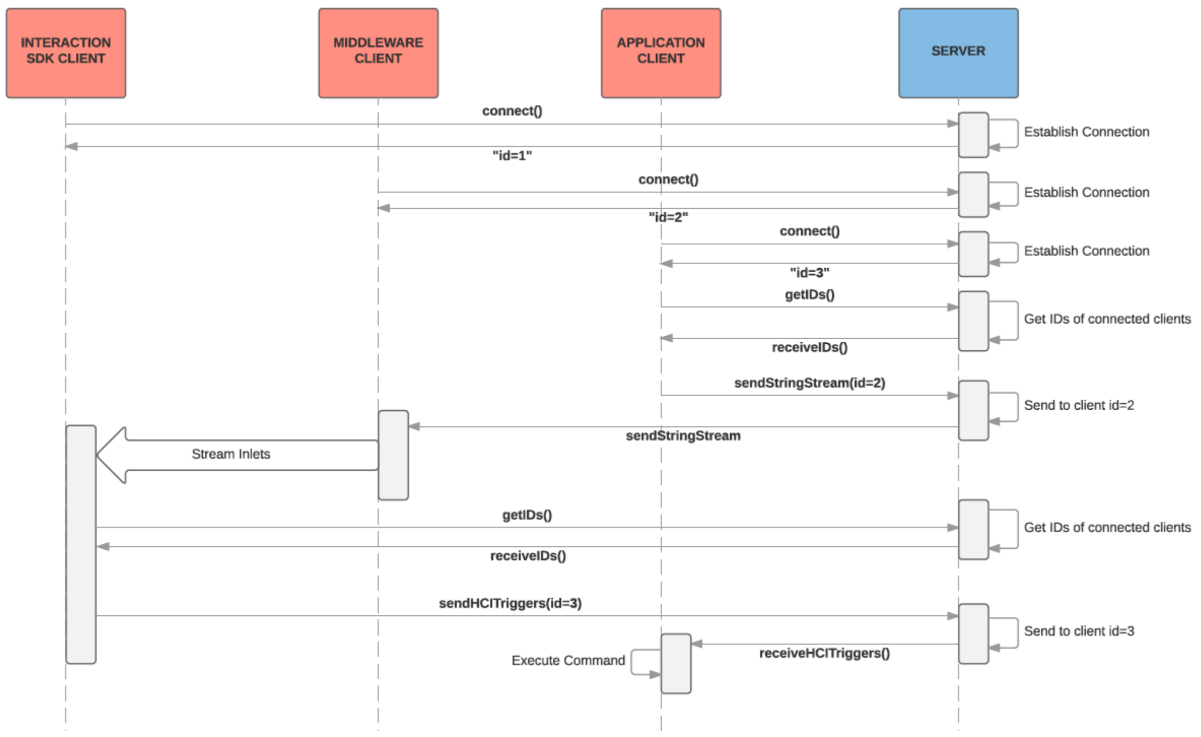


Figure 26: Typical flow of communication using messages in a client-server model

When each client is initialized it sends a connect command to the server and receives its ID from the server. Before sending a message to another client, the id of the receiver client must be requested from the server. An example response containing the list of clients connected to the server is the following.

```
{
  "connected_clients": [
    {
      "type": "webapp",
      "id": 3
    },
    {
      "type": "interactionSDK",
      "id": 1
    },
    {
      "type": "middleware",
      "id": 2
    }
  ]
}
```

If an application client with an ID=3 wants to send a message to the middleware (ID=2) it shall send the message to the id matching the type=”middleware”. A string stream can be pushed to the middleware with a message such as the following.

```
{
  "senderID": 3,
  "receiverID": 2,
  "timestamp": 1448465003,
  "message": {
    "numevents": 3,
    "string_stream": "ssvep_events",
    "events": [
      {
        "event": "application_init",
        "timestamp": 1448464782
      },
      {
        "event": "stimuli_start",
        "timestamp": 1448464801
      },
      {
        "event": "stimuli_stop",
        "timestamp": 1448464805
      }
    ]
  }
}
```

The middleware client will then parse the message and generate a string stream named "ssvep_events" containing 3 event markers.

The same procedure is to be followed when the Interaction SDK wants to send the HCI Triggers to the application client. An example message can be the following.

```
{
  "senderID": 1,
  "receiverID": 3,
  "timestamp": 1448467003,
  "message": {
    "type": "trigger_command",
    "command_id": 1
  }
}
```

7 Logical view of MAMEM architecture

The logical view of MAMEM architecture is presented in Figure 27. Our goal with this view is to summarize and present in a comprehensive manner how the different *Layers*, *Modules* and *Interfaces* fit together in the proposed architecture.

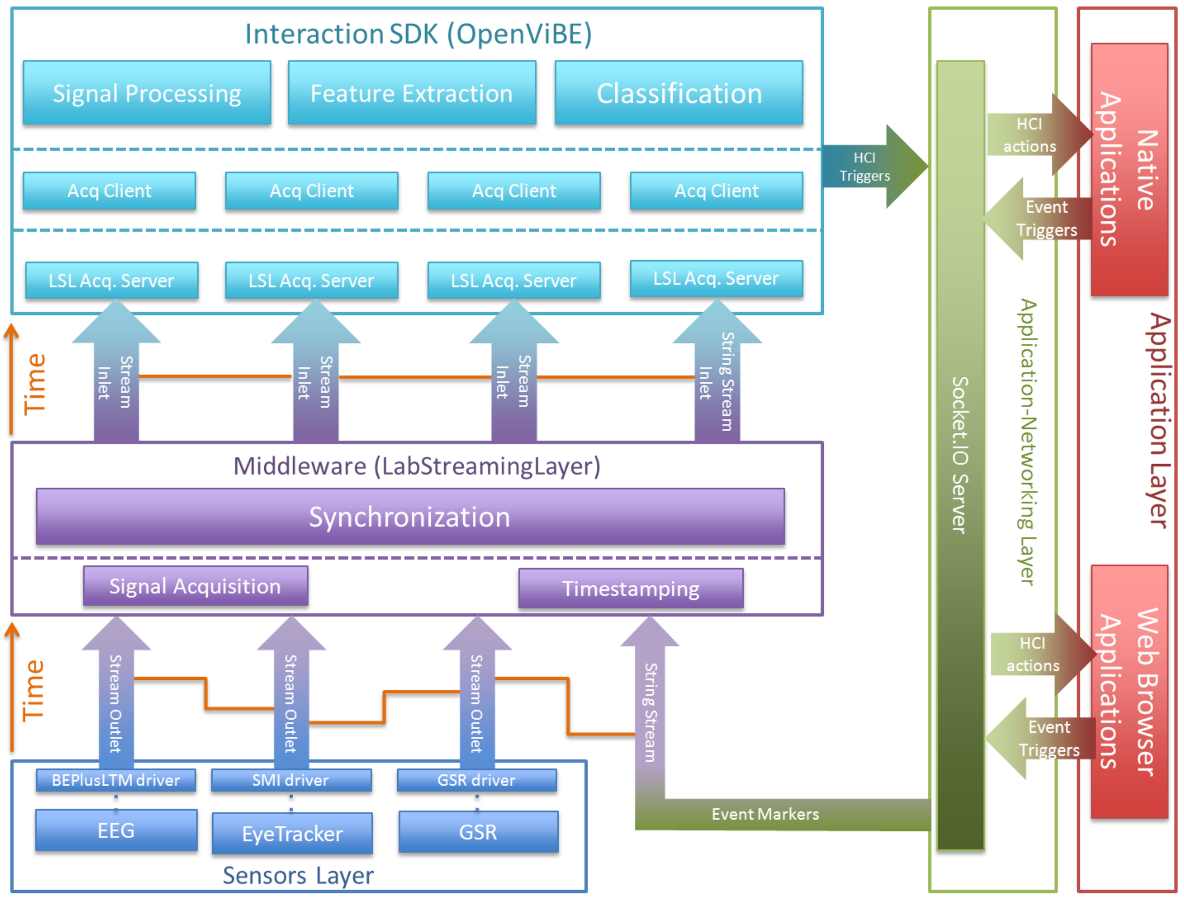


Figure 27: Overall MAMEM Architecture

This architecture is essentially an elaborated version of the high-level architecture that was presented in the DoA [5], see also Figure 4 of this document. There are four main *Layers* constituting our architecture, namely: a) *Sensors Layer*, b) *Middleware*, c) *Interaction SDK*, and d) *Applications Layer*. Each *Layer* incorporates a number of *Modules* that are responsible for undertaking the tasks that are necessary for MAMEM system to operate. Moreover, in order for one *Layer* to communicate with the other we define a set of *Interfaces.* These *Interfaces* specify how the information should be structured in order to pass from one *Layer* to another. Typically, the *Layers* are associated with *Input Interfaces* and *Output Interfaces* that determine the structure of the information that comes in and out of this *Layer.* Table 10 presents how the different *Layers* are associated with their *Modules* and *Interfaces* based on the proposed architecture. In addition, for each *Layer*, *Module* and *Interface* we provide a reference to the Section of this document (or another deliverable) that elaborates on its details.

Layer	Modules	Input Interface	Output Interface

Sensors Layer (Deliverable 2.1)	BEPlusLTM driver (Section 3.3.1)	Low-level device driver (Deliverable 2.1)	Stream Outlet (Section 3.3)
	SMI driver (Section 3.3.2)		
	GSR driver (Section 3.3.3)		
Middleware (LabStreamingLayer) (Section 3)	Signal acquisition (Section 3.4)	Stream Outlet (Section 3.3)	Synched Stream Inlet (Section 3.3 & Section 3.6.1)
	Timestamping (Section 3.5.5)		
	Synchronization (Section 3.5.5)	String Stream Outlet (Section 3.6.2)	Synched String Stream Inlet (Section 3.6.2)
Interaction SDK (OpenViBE) (Section 4)	LSL Acq. Server (Section 4.3.1)	Stream Inlet from LSL (Section 4.3.1)	Signal streamed on the network (Section 4.3.1)
	LSL Acq. Client (Section 4.3.1)	Signal from LSL Acq. Server (Section 4.3.1)	Signal streamed in OpenViBE boxes (Section 4.3.1)
	Signal Proc., Feat. Extrac., Classification (Section 4.3.2)	Signal from LSL Acq. Client (Section 4.3.1)	HCI Triggers (Section 6.2)
Application-Network Layer (Section 6)	Socket. IO Server (Section 6.1)	HCI Triggers (Section 6.2)	HCI Actions (Section 5.2)
		Event Triggers (Section 6.2)	String Steam Outlet (Section 3.6.2)
Application Layer (Section 5)	Native Application (Section 5)	HCI Actions (Section 6.2)	Change HCI status of front-end application (Section 5)
	Browser Application (Section 5)	Start and HCI process in front-end	Event Triggers (Section 6.2)

		application (Section 5)	

Table 10: Association of MAMEM's *Layers, Modules* and *Interfaces* based on the proposed Architecture

The first *Layer* is the ***Sensors Layer*** that has the role of capturing the signal from the sensor devices in order to push it further up in our architecture. Apart from the actual hardware (i.e. EEG recorder, Eye-tracker and GSR sensor), among the *Modules* of the *Sensors Layer* we also classify the *Drivers* (one for each hardware device) that are necessary to make the generated signals available for MAMEM's *Middleware*. These *Drivers* have been built using the SDK of each device (see Deliverable 2.1 [36]) so as to provide the signals in a structure suitable for our *Middleware*. This structure is essential the *Stream Outlet Interface* which specifies how to structure a stream so as to be compatible with MAMEM's middleware. The *Stream Outlet Interface* is the only *Output Interface* of the *Sensors Layer*, whereas as *Input Interface* we may consider the output of the low-level drivers that typically come along with the sensor devices (see Deliverable 2.1 [36]).

The second *Layer* is the ***Middleware***. The role of this *Layer* is to act as the mediator between the sensor devices and the rest of MAMEM's architecture. In other words, as long as a new sensor device can comply with the *Input Interface* of this middleware, the rest of the system should operate seamlessly. In the proposed architecture, the framework named *LabStreamingLayer* has been chosen to serve as MAMEM's middleware (see Section 3.1 . The *Middleware* consist of three *Modules*, namely *Signal Acquisition*, *Timestamping* and *Synchronization*. *Signal Acquisition* is the module responsible for receiving the signals from the *Sensors Layer*. The role of *Timestamping* is to add timestamps on the received signals. Finally, the most important *Module* of our *Middleware* is the *Synchronisation Module* that takes care of synchronizing the (originally de-synchronized) signals based on their timestamps (as well as other sensor-specific delays, see Section 3.5.5 As a result, the signals can be pushed further-up in MAMEM's architecture in a synchronized mode. There are two input and two output interfaces associated with the *Middleware*. The input interfaces are the *Stream Outlet* and the *String Stream Outlet*. This first refers to the structure that should be followed by the signal coming from the *Sensors Layer*. The second is a specific type of interface that allows our *Middleware* to receive strings at irregular frequencies, which are typically used to mark the beginning or the end of an event. In our architecture, we foresee the use of this interface as the means for the end-user application to communicate the point in time where a certain action or process should be initiated in the back-end. Finally, the output interfaces consists of the *Synched Stream Inlet* and *Synched String Stream Inlet*, which specify the structure of the signal and string streams that should be followed by another layer in order to receive information from the *Middleware*. The main difference compared to the signals received by the Middleware is that on its output the signals are synchronized.

The third *Layer* in our architecture is the ***Interaction SDK*.** The role of this *Layer* is threefold: a) communicate transparently with the *Middleware*, so as to receive the synchronized signals, b) implement an extensive list of processes and methods for translating the bio-signal into triggers for the HCI, and c) communicate these triggers with the front-end of our system. In the proposed architecture, the framework named *OpenViBE* has been selected to serve as the back-bone of MAMEM's Interaction SDK (see Section 4.1). OpenViBE consists of various different *Modules.* First, we should refer to the *Acquisition Server* which is the module responsible for receiving the *Signal and String Inlets* from our *Middleware*. This module should be instantiated as many times as the number of existing signal and string streams and ensures that these streams will made available to the processes and methods implemented within OpenViBE. Actually, in order to do this, all instantiated *Acquisition Servers* should be paired with an instance of an *Acquisition Client,* which is the other module classified under the *Interaction SDK Layer.* This module ensures that the incoming signal and string streams will be made available to the processing *Modules* of OpenViBE, such as *Signal Processing*, *Feature Extraction* and *Classification*. Finally, in terms of the associated interfaces we can distinguish between *External Input Interfaces*, *Internal Input Interfaces* and *Output Interfaces*. In the *External Input Interface* we can classify the *Signal and String Stream Inlet,* which is the structure of the streams that comes out of the *Middleware Layer*. In the *Internal Input Interfaces* (these do not appear in Figure 27 to maintain the clarity of the diagram) we can classify the *Signal Acquisition Server* that is used to pass the streams from the *Acquisition Server* to the *Acquisition Client* and the *Signal Acquisition Client* that is used to pass the signal from the *Acquisition Client* to the *Processing Modules*. Finally, in the Output Interfaces we classify the *HCI Triggers Interface*, which is essentially the structure of the messages that are communicated from the Interaction SDK to the *Application-Network Layer* and subsequently to the *Application Layer* in order for certain HCI commands to be executed in the front-end application.

The fourth *Layer* in our architecture is the ***Applications-Network Layer***. The role of this *Layer* is to handle the communication between the *Interaction SDK* and the front-end *Application Layer*. In particular, the *Applications-Network Layer* consists of one module namely, *Socket.IO Server,* which is intended to implement a server that will be able to communicate with a number of clients through network sockets. There are four different interfaces associated with this *Layer*. Among the *Input Interfaces* we can classify the *Event Triggers* and the *HCI Triggers*. The *Event Triggers Interface* is the structure of the messages that should be communicated by the end-user application to inform the back-end system that a certain task has started (e.g. the process of presenting the visual stimuli to the user has started and the back-end system should decide which of the flickering boxes is selected by the user based on his/her brain electrical signals). These messages will be subsequently formulated in a *String Stream Outlet* that will be passed on to the *Middleware*. The *HCI Triggers Interface* is the structure of the messages that should be communicated from the *Interaction SDK* to the *Applications-Network Layer* in order to pass on the information about the output of a certain signal analysis tasks (e.g. continuing from the previous example, these messages should inform the *Applications-Network Layer* that the flickering box that has been selected by the user is the one on the upper-left part of the screen). Finally, the output interfaces of the *Applications-Network Layer*, consists of: a) the *HCI Actions Interface*, which specifies the structure of the information that will be received by the native or web browser application and be translated in commands for the interface; and b) The *String Stream Outlet Interface*, which takes care of translating the event triggers into a string stream outlet suitable for our *Middleware*.

The fifth and final layer of our architecture is the **Application Layer**. This *Layer* is used to represent the end-user applications that will be operated through the users' eyes and mind. It consists of two modules titled as *Native Application* and *Web Browser Application* that are used as containers for all different types of applications that will be developed in MAMEM. The *Interfaces* in this *Layer* are essentially identical with the ones presented in the *Applications-Network Layer* where the input/output property is reversed. Thus, the *HCI Action Interface* is now the input interface that determines the structure of the messages that will be translated into HCI commands, and the *Event Triggers Interface* is the structure of the messages that are used to mark the begin/end of a certain HCI-related task.

8 Conclusions

In this document we have described the proposed MAMEM architecture by specifying all different components involved, ranging from the sensor devices and the middleware, all the way to the interaction SDK and the communication with the end-user applications. The logic instruments that have been used to describe our architecture are *Layers*, *Modules* and *Interfaces. Layers* are used to denote the parts of our system that serve a different purpose. *Modules* are used to describe the core functionalities performed in each layer and the *Interfaces* are used to specify how the information flows from one layer to another. Throughout the document we have provided elaborated descriptions for one of these *Modules* and *Interfaces* and in Section 7 we have presented how the different component fit together.

MAMEM's architecture consists of five different *Layers* that distinguish between the *Sensors Layer*, the *Middleware*, the *Interaction SDK*, the *Application-Network Layer* and the *Applications Layer*. What is particularly interesting in the proposed architecture is that we have decided to use as back-bone for the Middleware and the Interaction SDK, two existing frameworks namely, LabStreamingLayer and OpenViBE. This decision has been favoured for two main reasons: a) avoid replicating the development effort that has been already undertaken by the respective community, b) increase the impact of the new functionalities developed within MAMEM, since they will automatically reach a rather extended community. After carefully examining these two systems (and their competitors, see Sections 3 and 4), we have reached the conclusion that they can adequately cover the requirements derived from MAMEM objectives, and they feature the necessary level of modularity so as to extent them with MAMEM-specific functionalities.

Finally, it is important to make a special reference to the *Application-Network Layer* which is the layer used to handle the communications between the back-end of our system and the end-user applications. Although this layer adds some additional complexity in the implementation of a new functionality, it has been deemed necessary to ensure that MAMEM system will be able to communicate with all different type of front-end applications, independently on whether are running as native applications or through a web browser.

9 References

[1] "Realtime Web Analytics With no Sampling," 2015. [Online]. Available: https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0&qpsp=201&qpnp=1&qptimeframe=M.

[2] "Emotiv EPOC," Emotiv, 2015. [Online]. Available: https://emotiv.com/epoc.php.

[3] "TIOBE Index," October 2015. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[4] A. S. Tanenbaum and v. S. Maarten, Distributed Systems : Principles and Paradigms, Prentice Hall, 2002.

[5] "MAMEM Project Description of Actions – PartB," 2015. - Readable form (requires authentication): http://mklab.iti.gr/mamem/images/3/32/PartB_MAMEM.pdf

[6] "Brain Products GmbH," 2015. [Online]. Available: http://www.brainproducts.com/. [Accessed October 2015].

[7] "EYE-EEG plugin," 2015. [Online]. Available: http://www2.hu-berlin.de/eyetracking-eeg/.

[8] "EEGLAB," 2015. [Online]. Available: http://sccn.ucsd.edu/eeglab/.

[9] "BIOPAC ACQKNOWLEDGE SOFTWARE," 2015. [Online]. Available: http://www.biopac.com/product/acqknowledge-software/.

[10] "Open Vibe - Software for Brain Computer Interfaces and Real Time Neurosciences," Inria, 2015. [Online]. Available: http://openvibe.inria.fr/.

[11] "Virtual Reality Peripheral Network - Official Repo," 2015. [Online]. Available: https://github.com/vrpn/vrpn/wiki.

[12] "Labstreaminglayer," 2015. [Online]. Available: https://github.com/sccn/labstreaminglayer.

[13] "Streaming Layer API," 2015. [Online]. Available: https://github.com/sccn/labstreaminglayer/wiki.

[14] "LSL - Basic Operation," 2015. [Online]. Available: https://code.google.com/p/labstreaminglayer/wiki/BasicOperation.

[15] "Rolling Shutter vs. Global Shutter," 2015. [Online]. Available: http://www.qimaging.com/ccdorscmos/triggering.php.

[16] "Sparkfun - Logic Levels," 2015. [Online]. Available: https://learn.sparkfun.com/tutorials/logic-levels/introduction.

[17] "Logic Levels - TTL Logic Levels," 2015. [Online]. Available: https://learn.sparkfun.com/tutorials/logic-levels/ttl-logic-levels.

[18] "Cedrus StimTracker," 2015. [Online]. Available: http://cedrus.com/stimtracker/.

[19] B. Guinot, "Solar time, legal time, time in use," in Metrologia, Volume 48, 2011, pp. S181-S185.

[20] D. L. Mills, Computer Network Time Synchronization: The Network Time Protocol, Taylor & Francis, 2010.

[21] "Stream Header Synchronization Parameters," 2015. [Online]. Available: https://github.com/sccn/labstreaminglayer/wiki/TimeSynchronization.wiki#stream-header-synchronization-parameters.

[22] "Labstreaminglayer Wiki," 2015. [Online]. Available: https://github.com/sccn/labstreaminglayer/wiki.

[23] Hend Suliman Al-Khalifa, Remya P.G., Eye Tracking and e-Learning Seeing ThroughYour Students' Eyes, eLearn Magazine. ACM Publication, June 2010.

[24] Sistem integrat de asistare pentru pacienŃi cu afecŃiuni neuromotorii severe (RO), PNCDI 2, http://telecom.etc.tuiasi.ro/telecom/staff/rbozomitu/asistsys/

[25] Lupu R.G., Ungureanu F., Bozomitu R., Mobile Embedded System for Human Computer Communication in Assistive Technology. Proceedings IEEE ICCP 2012, Cluj-Napoca, Romania, 209−212, August 2012.

[26] Lupu R.G., Ungureanu F., Siriteanu V., Eye Tracking Mouse for Human Computer Interaction. The 4th IEEE International Conference on E-Health and Bioengineering - EHB 2013, Iaşi, Romania, November 21-23, 2013.

[27] Jacob, R. J. K. 1993. Eye Movement-Based Human-Computer Interaction Techniques: Toward Non-Command Interfaces. http://www.eecs.tufts.edu/~jacob/papers/hartson.pdf

[28] Zhai, S., Morimoto, C. and Ihde, S. 1999. Manual And Gaze Input Cascaded (MAGIC) Pointing. Proc. of the SIGCHI conference on human factors in computing systems: the CHI is the limit, pp. 246-253.

[29] Lupu, Robert Gabriel, and Florina Ungureanu. "A survey of eye tracking methods and applications." Bul Inst Polit Iaşi (2013): 71-86.

[30] Jönsson, Erika. "If looks could kill–an evaluation of eye tracking in computer games." Unpublished Master's Thesis, Royal Institute of Technology (KTH), Stockholm, Sweden (2005).

[31] Smith, J. David, and T. C. Graham. "Use of eye movements for video game control." Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology. ACM, 2006.

[32] Jacob, R. J. What You Look at is What You Get: Eye Movement Based Interaction Techniques. In CHI '90, 1990, pp. 11-18.

[33] Tanriverdi, V., Jacob, R. J. K. Interacting with Eye Movements in Virtual Environments. In CHI '00 Proceedings (2002), ACM, pp. 265-272.

[34] Core Transport Library of LabStreamingLayer, source: https://www.youtube.com/watch?v=Y1at7yrcFW0.

[35] Hardware supported by LabStreamingLayer, source: https://code.google.com/p/labstreaminglayer/wiki/SupportedDevices

[36] D2.1 - Prototype modules implementation for signal capturing, MAMEM Consortium, November 2015. url (requires authentication to MAMEM wiki): http://mklab.iti.gr/mamem/images/f/fd/D2.1_Sensor_configuration_and_signal_capturing_final.pdf

[37] Available Tools for Brain Computer Interfaces, Christian A. Kothe Swartz Center for Computational Neuroscience, University of California San Diego, source: https://www.youtube.com/watch?v=rpA7uGa5nDM&index=7&list=PLbbCsk7MUIGcO_lZMbyymWU2UezVHNaMq

[38] The BioSig Project, http://biosig.sourceforge.net/

[39] BCI2000 – SCHALK LAB, http://www.schalklab.org/research/bci2000

[40] BCI Lab Group, http://about.bci-lab.info/home

[41] Y. Renard, F. Lotte, G. Gibert, M. Congedo, E. Maby, V. Delannoy, O. Bertrand, A. Lécuyer, “OpenViBE: An Open-Source Software Platform to Design, Test and Use Brain-Computer Interfaces in Real and Virtual Environments”, Presence : teleoperators and virtual environments, vol. 19, no 1, 2010

[42] OpenViBE, Tutorial – Level 1 – The most basic OpenViBE setup, source: http://openvibe.inria.fr/tutorial-the-most-basic-openvibe-setup/

[43] OpenViBE, List of processing boxes, source: http://openvibe.inria.fr/documentation/unstable/Doc_BoxAlgorithms.html

[44] OpenViBE, Implementing a Signal Processing Box, source: http://openvibe.inria.fr/tutorial-2-implement-algorithm-and-use-it-in-boxes/

[45] OpenViBE, Handling Messaging Between Boxes, source: http://openvibe.inria.fr/box-messaging-for-developers/

[46] OpenViBE, Making Boxes available in the designer, source: http://openvibe.inria.fr/introduction-algo-boxes/

[47] OpenViBE, Supported Acquisition Systems, source: http://openvibe.inria.fr/supported-hardware/

[48] Wolpaw J, Birbaumer N, McFarland DJ, Pfurtscheller G, Vaughan TM, “Brain-computer interfaces for communication and control.“ Clin Neurophysiol. 2002 Jun;113(6):767-91.

[49] Guger, C., et al. (2009). How many people are able to control a P300-based brain–computer interface (BCI)? Neuroscience Letters, 462, 94–98.

[50] Intendix (2010), source: http://www.intendix.com

[51] TOBI (2013), Tools For Brain Computer Interaction, ICT Project, source:

http://www.tobi-project.org/

[52] Graz BCI Lab (2013), source: http://bci.tugraz.at

[53] WebSocket, source: http://www.websocket.org/

[54] Socket. IO Model, source: http://socket.io/

[55] Socket.IO example, source: https://github.com/socketio/socket.io-client-cpp

[56] NodeJS, soruce: https://nodejs.org/en/

[57] C. Guger, A. Schlögl, C. Neuper, D. Walterspacher, T. Strein, and G. Pfurtscheller, “Rapid prototyping of an EEG-based brain-computer interface (BCI),” IEEE Trans. Neural Syst. Rehab. Eng., vol. 9, no. 1, pp. 49–58, 2001

[58] G. Pfurtscheller, R. Leeb, C. Keinrath, D. Friedman, C. Neuper, C. Guger, and M. Slater, “Walking from thought,” Brain Res., vol. 1071, no. 1, pp. 145–152, 2006.

[59] N.J. Hill, T.N. Lal, M. Schroder, T. Hinterberger, B. Wilhelm, F. Nijboer, U. Mochty, G. Widman, C. Elger, B. Scholkopf, A. Kubler, and N. Birbaumer, “Classifying EEG and ECoG signals without subject training for fast BCI implementation: Comparison of nonparalyzed and completely paralyzed subjects,” IEEE Trans. Neural Syst. Rehab. Eng., vol. 14, pp. 183–186, June 2006.

[60] N. Weiskopf, K. Mathiak, S.W. Bock, F. Scharnowski, R. Veit, W. Grodd, R. Goebel, and N. Birbaumer, “Principles of a brain-computer interface (BCI) based on real-time functional magnetic resonance imaging (fMRI),” IEEE Trans. Biomed. Eng., vol. 51, pp. 966–970, June 2004.

[61] S.-S. Yoo, T. Fairneny, N.-K. Chen, S.-E. Choo, L.P. Panych, H. Park, S.-Y. Lee, and F.A. Jolesz, “Brain-computer interface using fMRI: Spatial navigation by thoughts,” Neuroreport, vol. 15, no. 10, pp. 1591–1595, 2004.

[62] F. Matthews, B.A. Pearlmutter, T.E. Ward, C. Soraghan and C. Markham, "Hemodynamics for Brain-Computer Interfaces," in Signal Processing Magazine, IEEE , vol.25, no.1, pp.87-94, 2008

[63] Smartbox, http://thinksmartbox.com

[64] Tobii Dynavox, www.tobiidynavox.com

[65] Tobii AB, www.tobii.com

[66] Eyetech Digital Systems, www.eyetechds.com

[67] LC Technologies, www.eyegaze.com

[68] Prentke-Romich Company, www.prentrom.com

[69] SensoMotoric Instruments, www.smivision.com

[70] Visual Interaction myGaze, www.mygaze.com

[71] The Eye Tribe, http://theeyetribe.com

[72] OptiKey, www.optikey.org

[73] The EyeWriter Project, www.eyewriter.org

[74] Vision Key, www.eyecan.ca

[75] EBNeuro BEPlusLTM, http://www.ebneuro.biz/en/neurology/ebneuro/galileo-suite/be-plus-ltm

[76] SMI iViewREDN, http://www.smivision.com/en/gaze-and-eye-tracking-systems/products/red250-red-500.html

[77] Shimmer3 GSR+, http://www.shimmersensing.com/shop/shimmer3-wireless-gsr-sensor

[78] Emotiv EPOCH, https://emotiv.com/product-specs/Emotiv%20EPOC%20Specifications%202014.pdf

[79] myGaze Assistive, http://www.mygaze.com/fileadmin/download/Tech_Specs/mygaze_assistive_2_techspecs.pdf

[80] EGI 300 Geodesic EEG System (GES 300), http://www.egi.com/clinical-division/clinical-division-clinical-products/ges-300

[81] SR Research, http://www.sr-research.com/

[82] SCHAUGENAU, http://schaugenau.west.uni-koblenz.de

[83] GitHub, https://github.com/

[84] Named pipes, https://en.wikipedia.org/wiki/Named_pipe

[85] Memory-mapped file, https://en.wikipedia.org/wiki/Memory-mapped_file

[86] Unix signal, https://en.wikipedia.org/wiki/Unix_signal

[87] Sockets, https://en.wikipedia.org/wiki/Network_socket

[88] Socket-IO-peer-to-peer, http://socket.io/blog/socket-io-p2p/

[89] WebRTC, http://www.webrtc.org/