Unlock: A Framework for Programming Brain-Computer Interfaces

James Percent
james@shift5.net

March 1, 2014

Contents

1   Overview

Our goal is to develop BCI technology to help individuals suffering from locked-in syndrome [1] communicate. Locked-in syndrome is characterized by an almost complete inability to communicate. With current technology, unfortunately, we cannot achieve this goal.

We think if more smart people are able to collaborate, then we have a better shot at accomplishing this goal. Therefore, our strategy is to focus on providing a programming framework that makes it easier for researchers and engineers to build BCI-based systems. To this end, we employ a few simple ideas.

We encapsulate as much as possible, which is to say that we separate concerns. We'll talk more about this later, but the basic idea is that you should not need to know every corner of Unlock to create a new decoding algorithm or add a language model or add a new acquisition device.

These could be different people with different skill sets, and we want them to be able to dive into their area and be productive quickly. One of the challenges with BCI technology is that it is inherently cross-disciplinary; therefore, we want to lessen this burden as much as possible.

To support this goal, we need great documentation. The documentation needs to be encapsulated too. You should be able to get a high-level idea of how the whole system works and also locate details about your specific area of interest without arduous effort.

Finally, automate whenever possible; extra batteries included. We want researchers and engineers to be able to get going quickly. Learning about all the dependencies and taking a bunch of manual steps can be overwhelming. Especially if someone just wants to hack out a quick idea.

In summary, the goal is to give you a great platform to hack out an idea.

The rest of this document is structured as follows. First we do a quick technical introduction. After that, we discuss how to install and use Unlock, from a user perspective. Next, we cover both some of the internals from a high-level. Finally we close with a example.

Development of the Unlock framework is funded by the Adaptive Brain-Computer Interactions (ABCI) Capstone Project [4].

2   Technical Introduction

The infrastructure Unlock provides can be logically separated into two distinct layers: the application layer and the BCI layer. These layers can in turn be further decomposed.

The BCI layer is defined by the signal-processing chain [2]. The signal-processing chain includes signal acquisition and reconstruction, feature extraction and decoding, and command generation [3]. Many BCI systems also include a stimulation component, which is actually processed at the application layer; however, logically, the stimulation component is part of the BCI.

For example, a paradigm based on Visually Evoked Potentials (VEP), flashes a graphical sequence at the user, which is subsequently interpreted in the signal-processing chain [6]. But the actual flashing is handled at the application layer.

We'll talk in more detail about the BCI layer in Section 4.2, but for now there is one important take away. At each point in the pipeline we just discussed, the system is designed so that you can easily decorate, replace or otherwise manipulate the data flow. Each step in the pipeline is encapsulated under an interface contract.

The application layer is model-view-controller (MVC) **??** inspired. Don't worry too much if you're not familiar with MVC. It's just a logical framework for structuring graphical user interfaces (GUIs).

The basic idea is that each application consists of three types of objects: models, views and controllers. The controllers accept user input (in this case BCI commands) and pass the inputs to the correct models. Models handle state transformation (the business logic if you will). Finally, views observe the state objects (models) and render themselves based on the current state of the world.

If you think about it, this also creates a pipeline in the application layer. The take away here is that these components are also encapsulated under interfaces.

Figure **??** depicts the main components of Unlock.

3 Installation

In the next three sections we describe the installation procedures for Unlock installation on Windows, Linux and Mac OSX.

3.1 Windows

Many signal acquisition devices work exclusively with Windows. Additionally, many of the potential users of Unlock may only have access to Windows computers. These circumstances led to Windows being the first platform supported by Unlock.

To make installation on Windows as easy as possible, we decided a binary, clickable installer was the solution that would best suite the community. There were many ways to proceed, but, ultimately, we decided to write the installer in Go [9].

The Unlock installer is not much more than a package manager for handling dependencies. Unlock, not unlike many other frameworks, has many dependencies. These dependencies include:

- Avbin10;

- Flask-0.10;

- Numpy 1.7.1;

- Psycopg2 2.5.1;

- Pyaudio 0.2.7;

- Pyglet 1.2alpha;

- Pyserial 2.6;

- Python 3.3.2;

- Pywin32 2.1.8;

- Scikit-learn 0.14.1;

- Scipy-stack 13.10.11;

- Scons 2.3.0;

- SQLAlchemy 0.9.0b1;

- Visual C++ redistribution 2010.

There are additional dependencies based the hardware acquisition device in use. Unlock supports three acquisition devices. The National Instruments DAQ, Neural Electronics Enobio, and gtec MOBILab.

Unfortunately, at the time of this writing, Unlock's C++ layer is built as a single dynamic-link library (dll). This means all of these dependencies need to be available for Unlock to run.

Fortunately, Unlock manages all but the NI DAQ driver code. The NI DAQ dependencies are quazi installed by Unlock, but may require some manual effort.

The Unlock installer will start the NI DAQ downloader, which should kick off the installer after the download and extraction complete. But it runs asynchronously to the Unlock installer; the Unlock installer exists without knowing if the NI DAQ installer completed successfully or not; if something

happens, go to [12] and follow the instructions for downloading and installing the NI DAQ.

Most of the dependencies are Python packages. We played with standard Python packaging systems, such as Pip [11] for installing these, but, unfortunately, at the time of this writing, none of these support transitive closure over dependencies. This makes packaging imperative and error prone.

Luckily we found [10]; it has most Python packages as Windows binaries that include their dependencies. Where ever possible we use the binary package from [10].

We are watching Pip and may revisit using the Pip packages in the future. That said, Pip would still be wrapped by the Go executable.

3.1.1 Developer Installation

At this time only developer-based installation is supported. A developer-based install requires a few steps:

- install Git Bash;

- setup a Github account;

- clone the Github repo;

- run the binary installer for your architecture (only x86-64 is supported right now).

- install NI DAQ

- edit Unlock's config

- run Unlock to test

Go to [15] for Git Bash download and installation instructions.

After Git Bash is setup we need to create a Github account [18] and follow the instructions for setting up an ssh key [19]. Next, fork the repo.

After we have your Github account setup we can clone the repo. This takes a few minutes because we committed all of Unlock's dependencies to the repo. *Stark* this is generally bad form. But we are OK with bad form for two related reasons: 1) we are poor and do not have a real server, 2) we care more about keeping everything together more than repo download speed.

To clone the repo run the following command from the Git Bash command.exe:

```
$ git clone git@github.com:jpercent/unlock.git
```

Note, git@github.com:jpercent/unlock.git is my fork of the project. If you want to commit to my fork that's fine, send me email and I'll add you. But it's probably better to commit to your own fork. If you want to push changes upstream, then you make a pull request.

Then change directory to the repo. To do so type:

```
$ cd unlock/
```

Then run the installer for a development installation:

```
$ golang/bin/install-win-x86-64.exe -dev=true -repo=`pwd`
```

Note, the quotes around pwd are back ticks; they tell Bash to execute the command that gets the current working directory. There should almost immediately be several installers that will ask you questions. Please accept the defaults on everything.

Watch the task bar. Some versions of Windows will bring up access control dialogs; sometimes they come up in the background. Additionally, at least one of the installers has a dialog that comes up asynchronously and in the background, so keep your eyes peeled for dialog boxes in the background. They should be flashing on the task bar.

Additionally, AVbin installer puts the avbin.dll in the wrong directory; the installer tries to move it, but you might have to do it manually as the administrator. To fix the issue go to C:

Windows

SysWOW64. If there is no avbin.dll, then copy the one in C:

Windows

System32 to SysWOW64.

That should be all there is to it. To test it run the following command:

```
$ export PYTHONPATH='pwd'
$ cd unlock/
$ /c/Python33/python.exe unlock_runtime.py
```

If you have an issue, please send me an email directly. Please include the file named unlock-install.log in the email and a description of the real-time symtom(s).

3.2 Linux

3.3 Mac OSX

4 Internals

4.1 configuration

Unlock builds most components using an "inversion of control" container. Most people refer to this, nowadays, as dependency injection [16]. What it means is that object construction is handled by a container that can be configured to wire up various of configurations based on the contracts/interfaces objects support. This paradigm of configuration management has been around a long time, but became widely known when the developers of J2EE did not know about it.

The important point, from a mechanical perspective, is that objects rarely create other objects in their own constructors. In Python each file is referred to as a module. And each package is defined at the directly level. Each unlock package has it's own factory that creates objects. The UnlockFactory creates and wires up the configuration of an unlock instance. What system is

created depends on the values set in the configuration file passed to Unlock. To understand how dependency injection works let's look at a small example:

```
"dashboard" : {
        "main" : true,
        "deps" : {
            "stimulation" : "quad_ssvep",
            "decoder" : "harmonic_sum",
            "controllers" : ["gridspeak", "time_scope", "frequency_scope",
                                       "gridcursor", "fastpad"]
        }
    },

    "signal" : {
        "singleton" : 0,
        "name" : "random"
    },

    "window" : {
        "singleton" : 1,
        "name" : "pyglet",
        "args" : {
            "fullscreen" : true,
            "fps" : false,
            "vsync" : false
        }
    },

    "harmonic_sum" : {
        "deps" : {
            "buffering_decoder" : "fixed_time_buffering_decoder",
            "threshold_decoder" : "absolute_threshold_decoder"
        },

        "args" : {
            "fs" : 256,
            "trial_length" : 3,
            "n_electrodes" : 8,
            "targets" : [12.0, 13.0, 14.0, 15.0],
```

```
        "target_window" : 0.1,
        "nfft" : 2048,
        "n_harmonics" : 1
    }
}
```

There are 4 elements here: "dashboard", "signal", "window" and "harmonic_sum". Here we are using JSON as the configuration language.

Each object that has an attribute named "singleton" is created automatically by the UnlockFactory and is available to any of the other objects listed in the config file by it's name. Note that the name is not the "name" attribute; the "name" attribute is actually the type, which it probably should be changed to. The order they are created in depends on the value of the "singleton" attribute - this allow us to support a dependency hierarchy. For example, the "window" singleton, which is of type pyglet_window, must be created after the "signal" singleton, which is of type random. I may remove this dependency hierarchy in favor of transitive closure over dependencies, but this was more simple to get started. For singletons a name attribute must be supplied; it corresponds directly with an UnlockFactory method - i.e. there is an unlock factory method named random, and 1 named pyglet_window.

Objects with the "singleton" attribute are created later. Note that the "dashboard" object has "main" set to true. This is the only none singleton object that will be created independently. Everyother object that gets created, must have a dependency off of the main object. For example, the "dashboard" lists "harmonic_sum" inside its "deps" json object. Primitive arguments are passed to a factory method using the "args" json object and complex objects are passed using the "deps". I may merge these together and make the factory a bit smarter, but this was simpler to start and that's how I learned to program (make it simple, make it work, then optimize).

4.2 BCI Layer

In Section 1 we defined the signal-processing chain to be signal acquisition and reconstruction, feature extraction and decoding, and command generation.

4.2.1 Stimulation

4.2.2 Acquisition

Most of the signal acquisition code is written in C++. The C++ layer is dynamically linked to the Python run-time using Boost Python [7]. The C++ signal acquisition module exports a single polymorphic interface inspired by UNIX device files, and several factory methods for creating the signal acquisition infrastructure.

Unlock supports two modes of acquisition: blocking acquisition and non-blocking acquisition. Blocking acquisition is is simply a thin wrapper around the device interface. It will block on a I/O descriptor until a write is available.

We will discuss Unlock's application layer in detail in Section 4.3. But at this time it is important to understand one aspect. The application layer is essentially a graphics event loop; looping through the application layer code at each screen refresh. To acquire the signal on the same thread as the graphics event loop is problematic: blocking on each screen refresh can cause all kinds of problems. In fact, blocking-based acquisition is only supported for device testing purposes.

To acquire the signal without blocking, Unlock supports a non-blocking acquisition mode. In non-blocking mode, Unlock creates a platform independent C++ thread that manages and buffers the signal data. The graphics event loop synchronizes with the signal acquisition thread using a lock-free algorithm. The method makes use of a ring buffer and processor level atomic instructions to coherently reserve blocks of virtual memory addresses across cores.

4.2.3 Decoding

4.2.4 Command Generation

4.3 Application Layer

The Unlock application programming layer is a model-view-controller (MVC) [5, ?] inspired framework for graphical user interface (GUI) development. Commands flow through the BCI layer to a control layer that interprets the state changes and sends the changes to visualization module.

4.4 Building Unlock for Development

4.5 Example

References

[1] Bauer, G. and Gerstenbrand, F. and Rumpl, E. Varieties of the locked-in syndrome. Journal of Neurology. 1979.

[2] Sippi, C., Sippi, P., Computer Dictionary and Handbook. 1972.

[3] He B., Gao S., Yuan H., and Wolpaw, J. Neural Engineering, Chapter 2, BrainComputer Interfaces. 2013.

[4] CELEST Capstone Project 1: Adaptive Brain-Computer Interactions. http://celest.bu.edu/about-us/capstone-projects/adaptive-brain-computer-interactions.

[5] Reenskaug, T. A note on DynaBook requirements. Xerox PARC. 1979.

[6] OShea, R. P., Roeber, U., Bach, M. Evoked potentials: Vision. Encyclopedia of Perception. 2010.

[7] Boost Python. http://www.boost.org/doc/libs/1_54_0/libs/python/doc/.

[8] Unlock. http://github.com/jpercent/unlock.

[9] Go. http://golang.org.

[10] Christoph Gohlke. http://www.lfd.uci.edu/ gohlke/pythonlibs/. Laboratory for Fluorescence Dynamics, University of California, Irvine.

[11] PIP. https://github.com/pypa/pip.

[12] NI DAQ. http://www.ni.com/dataacquisition/nidaqmx.htm.

[13] Eugster, P. Felber, P. Guerraoui, R. Anne-Marie Kermarrec, A. The many faces of publish/subscribe. ACM Computing Surveys (CSUR) Surveys Homepage table of contents archive. 2003.

[14] Gtec MOBILab. http://www.gtec.at/Products/Hardware-and-Accessories/g.MOBIlab-Specs-Features

[15] Git Bash. http://git-scm.com/download/win.

[16] Fowler, M. Fowler, Martin. Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html. 2003.

[17] Observer pattern. https://en.wikipedia.org/wiki/Observer_pattern

[18] Github. https://github.com.

[19] Github SSH key Generation. https://help.github.com/articles/generating-ssh-keys.