TOBI iD

Definition, implementation and scenarios

Michele Tavella

michele.tavella@epfl.ch

September 27, 2012

Contents

1 Introduction

The TOBI iD is designed by TOBIs WP5 and WP8 members, and its first implementation is maintained by EPFL. iD enables modules in the distributed hBCI design to exchange high-level events. Events allow all the distributed modules to react once a certain event is raised. As today iD provides a way to encode/decode messages but specially it defines the rules in which the modules exchange high-level events. When designing TOBI interfaces, the consortium has to:

1. Ensure compatibility of the TOBI interfaces with the existing BCI systems
2. Facilitate the use of the APIs and ensure flexibility in the hBCI design
3. Provide solutions that match the way the BCI community expects a BCI to work

From this perspective, other TOBI interfaces as the signal transmission iA or the classifier output definition and transmission iC require less effort because they implement point-to-point communication mechanisms that are limited within a specific subset of modules in the hBCI pipeline. With iD things change a bit, because the interface has to handle an all-to-all communication schema that requires the introduction of time information within the messages. For this reason the consortium agreed in moving towards a bus-like solution where the events from one module are broadcasted to all the other modules via an iD server. iD is available as a C++ library (Section 4.3) and as a MEX interface (Section 4.4).

Ongoing work

This document has not to be considered final due to the fact that the TOBI consortium is still in the procedure of defining the core functionalities of iD.

1.1 Disclaimer

The goal of this document is to make users familiar with hBCI designs including iD. The document is written in an easy format that should be familiar to most of the BCI community, thus not only addressing the hBCI-ninjas within the TOBI consortium or the code-monkeys in your lab. This document has to be considered an evolving text that will be continuously updated. Although it contains some references and examples based on libtobiid and mextobiid, this document does not replace the Doxygen or Matlab documentation shipped with libtobiid and mextobiid respectively.

1.2 Acknowledgment

This work is supported by the European ICT Programme Project FP7-224631, TOBI: Tools for Brain-Computer Interaction. This document only reflects the authors' views and funding agencies are not liable for any use that may be made of the information contained herein.

2 iD and the hBCI

Describing iD without defining a typical usage scenario would consist in replicating the API documentation provided with the library. As mentioned in Section 1, it is not possible not to consider the timing information when describing iD. For this reason it is necessary to introduce the concept of data frames. Biological signals are acquired by an acquisition module (i.e. the TOBI SignalServer) and sent via network to the feature extraction/classifiers modules. Each data frame is incrementally labeled with a corresponding frame number. When a module receives an input (i.e. a data frame via iA) and produces an output (i.e. a probability via iC), the output is labeled with the same frame number of the input.

By looking at Figure 1 and 2, we could simply say that in the hBCI pipeline(s), each module "passes" the frame number from left to right, following the direction pointed by the horizontal arrows. In the aforementioned figures, a rectangle represents the iD bus to wich each module is connected. The iD connections are depicted using vertical bidirectional arrows. In fact, while in the hBCI pipeline the information flows in a single direction at precise time instants via iA, iB and iC (i.e. input received and output emitted), iD allows the hBCI modules to communicate by means of events at any timepoint. The iD messages are written on the iD bus and transmitted to all modules in the hBCI. So we could say that a module can send and received iD messages at any time, and not following the clock dictated by the acquisition. Furthermore it means that a module can raise (or receive) and event with sub-frame accuracy. [It is a little tricky here, pls consider it very preliminary. I will re-write it in the future. Maybe it would be useful to provide a set of "rules" followed by the hBCI.].

In the hBCI design, multiple biological signals are acquired, classified and finally used to control a device. In the simplest case, a biological signal (i.e. EEG) is used to classify SMRs. This constitutes the simplest scenario in which we can describe how iD works. Another scenario involves the use of multiple biological signals (i.e. EEG and EMG) or the fact that we try to classify different phenomena starting from a single biological signal (i.e. SMR and ErrP). Both cases require the hBCI to develop on multiple processing pipelines.

The two scenarios above are described in Section 2.1 and 2.2 respectively. For the sake of simplicity, we will assume that feature extraction and classification are always handled within the same module.

2.1 Single BCI pipeline

The case of a single BCI pipeline is the simplest one to introduce the principles behind iD. In this case, an EEG signal is acquired at 512Hz and small chunks of information are sent at 16 Hz to an SMR classifier. This means that the SMR classifier receives 16 packets of 32 samples per second. It also means that feature extraction and classification need to consume less than one cycle (1/16 seconds) in order for the pipeline to run in real-time. As stated in the introduction, the asynchronous nature of iD requires the reader to understand what happens, at each cycle, at the level of the modules taking part to the hBCI assembly. We will then define f_{hBCI} as the frequency at which the EEG acquisition emits data packets. We also define T_{hBCI} at the duration, in seconds, of a cycle. In this particular example: $f_{hBCI} = 16Hz$ and $T_{hBCI} = (16Hz)^{-1} = 0.0625s$. We will

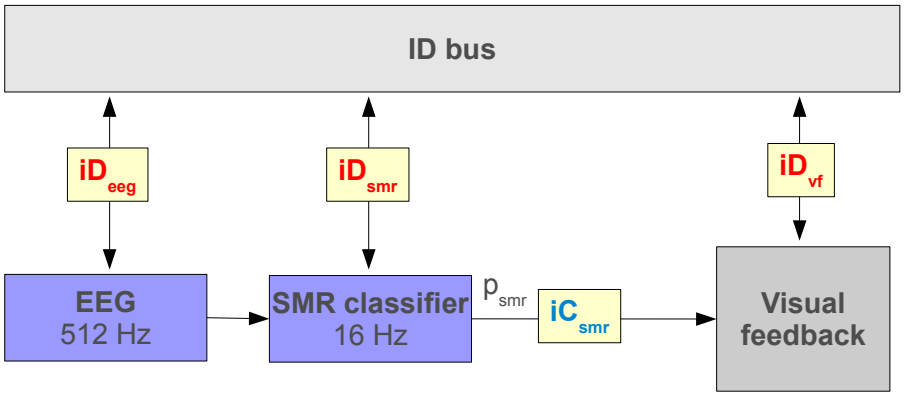

Figure 1:

also assume that each module checks for the presence of iD messages (vertical bidirectional arrows) continuously. The reader can imagine that each module is able to receive, parse and process iD events in multi-threaded fashion.

Let's now imagine that, every now and then, the visual feedback module raises an event, with the intent to notify the SMR classifier about its status. For example: the feedback module writes an event on the iD bus at frame F_n. All the other modules will receive immediately the event, reacting accordingly. Typically, the EEG module will write immediately the event to some kind of file. Similarly, the SMR classifier will react (i.e. changing the weights of an LDA classifier) so that at frame F_{n+1} the feedback module can receive and updated classifier output.

The importance of frame numbers

When a module in the hBCI assembly receives an iD message, it must compare the frame number of the iD message with the frame number of its iA or iC input. This is particularly crucial to access when and where a message was generated. If we assume that the whole hBCI pipeline can process data in real time (that is within $1/f_{hBCI}$ seconds), than any module can receive iD messages just from modules processing the very same EEG frame.

If the realtime assumption does not hold, things get more complicated. Let's consider a module M that at frame I receives an iD message from a module K processing frame J:

- $I = J$: M and K are processing the same frame, thus the realtime assumption holds (at least for these two modules)
- I ¿ J: M is $I-J$ frames ahead K. This means that a module is introducing a delay between modules M and K. It also means that module M precedes module K in the hBCI pipeline, or, according to Figure 1 and 2, M is on the left of K.
- I ¿ J: M is $J - I$ frames behind K. This means that a module is introducing a delay between modules K and M. It also means that module

M succeeds module K in the hBCI pipeline, or, according to Figure 1 and 2, M is on the right of K.

2.2 hBCI and multiple pipelines

The example in Section 2.1 describes how iD messages are exchanged in a simple BCI. The difference between a BCI and an hBCI has to be found in the multiple processing pipelines of the latter. Figure 2 shows an hBCI that acquires a single biological signal (EEG) to classify SMRs and ErrPs. In this case the EEG

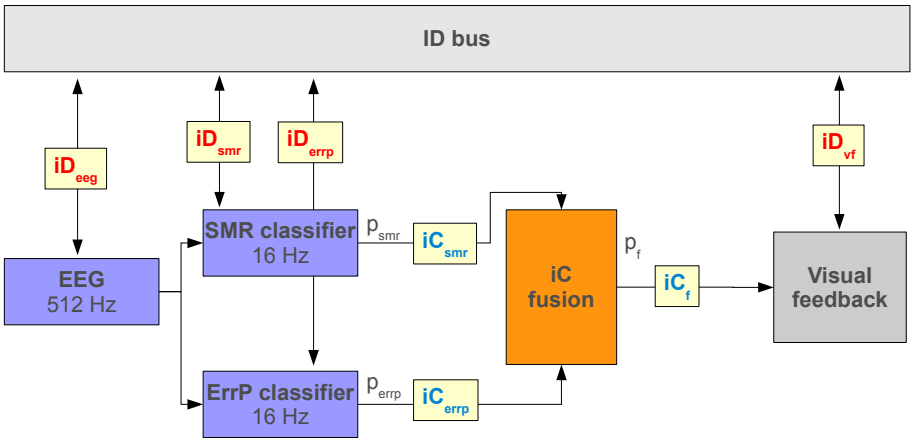

Figure 2: A prototypical hBCI in which the output of multiple pipelines (iC_{smr} and iC_{errp}) are fused in a single one (iC_f).

acquisition sends data frames both to the SMR and to the ErrP classifiers. As seen before, all modules can be interfaced with the iD bus, thus being able to send and receive messages. A core component of the hBCI is the fusion module. A fusion module receives multiple iC inputs and merge them in a single output, that is than transmitted to higher-level modules. With respect to iD and frame numbers, the fusion module carries the burden of aligning the input iC messages in time, thus to provide a consistent iC output. This becomes particularly critical when one of the input modules (i.e. SMR or ErrP) is delayed in time or when the input modules are producing iC outputs at different rates.

3 Structure of an iD message

iD messages are used to transmit events withing the hBCI assembly. Figure 3 depicts the attributes of an iD message. The attributes of an iD message are straight forward. Frame number and iD version are used to encode temporal information and libtobiid version number respectively. The main content of an iD message has to be found in the event code and in its family label. As today iD supports only Biosig and custom family types.

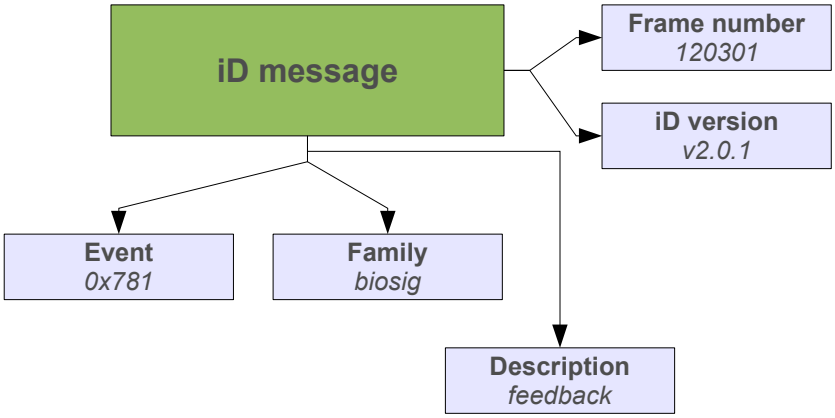

Figure 3: Internal structure of an iD message. The light blue boxes contain the attributes.

4 Implementation

4.1 Communication

As today iD is not shipped with a transport layer (i.e. TCP/IP, UDP) because different partners within the TOBI consortium use already different networking stacks. If you are not expert in coding and you want to evaluate iD, we strongly recommend to start with the MEX interface and eventually using jtcp or judp for sending data on the network.

4.2 Serialization

Up to now we always said that iD messages are exchanged between different modules. Still, we never addressed specifically how this messages are encoded and decoded for the transmission. Similarly to what has been said in Section 4.1 for the communication, different groups might require different solutions for encoding/decoding iD messages. For this reason the library provides a template class named *IDSerializer* on top of which is possible to implement any serialization method.

As today the consortium agreed on keeping the complexity of high-level communication as low as possible, thus choosing XML (using RapidXML as a back-end) for the serialization and de-serialization of the iD messages. Readers interested in this topic will find all the necessary informations in Section 5.

Some examples of serialized iD messages are available in Section .

4.3 libtobiid

The C++ library is distributed with a Doxygen API documentation and several examples. It can be downloaded from here: `http://`.

4.4 mextobiid

The MEX interface is distributed with the standard Mathworks Matlab documentation and several examples. It can be downloaded from here: `http://`.

5 Examples

An example coded in Matlab is hereby provided to illustrate how iD messages can be handled and how certain operation on frame numbers can be performed. The example resembles what has been described in Section 2.

The following are the main functional blocks within the provided example:

- Lines 19-32: configuration of iD as client
- Lines 42-45: configuration of an iD message as sent from a classifier module
- Lines 47-50: configuration of an iD message as sent from a feedback module
- Lines 52-55: configuration of a delayed iD message
- Lines 57-69: in these lines the iD events and the frame numbers are assigned. Furthermore, the messages are added to the client queue
- Lines 71-86: dequeuing of iD messages coming from modules that precede the simulated one
- Lines 88-103: dequeuing of iD messages coming from modules that succeed the simulated one.
- Lines 99-102: detection of iD messages transmitted from delayed modules

An XML message resulting from the iD serialization can be found in Section 5.1.

```
1  %   Copyright (C) 2010 Michele Tavella <michele.tavella@epfl.ch>
2  %
3  %   This file is part of the mexctobiid wrapper
4  %
5  %   The libndf library is free software: you can redistribute it and/or
6  %   modify it under the terms of the version 3 of the GNU General Public
7  %   License as published by the Free Software Foundation.
8  %
9  %   This program is distributed in the hope that it will be useful,
10 %   but WITHOUT ANY WARRANTY; without even the implied warranty of
11 %   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
12 %   GNU General Public License for more details.
13 %
14 %   You should have received a copy of the GNU General Public License
15 %   along with this program.  If not, see <http://www.gnu.org/licenses/>.
16
17 clear all;
18
19 % + Create a client
20 % Imagine the module is actually at frame 1000
21 % Imagine we are a "fusion" module, that is:
22 % - after the classifier
23 % - before the feedback
24 % A module before the fusion might have an higer frame number.
25 % A module after the fusion might have a lower frame number.
26 % I say, 'might', because I imagine the fusion might take longer
27 % than 1/f (where f is the processing frequency of the BCI) to
28 % process a given frame an move to the next.
29 % In theory, fusion and classification are really quick modules, so
30 % this scenario should not happen.
31 client = idasclient_new();
32 idasclient_setfidx(client, 1000);   % Client is at frame 1000
33
34 % + Fill up the message cue
35 % This simulates the fact that we receive iD messages
36 % from the broadcast. In reality, we should first read the messages
```

```
% from a socket and deserialize them.
% — messageC comes from the classifier (same frame number)
% — messageF comes from the feedback (one frame before)
% — messageD comes from a buggy module that takes more than 1/f to process the
%   current frame.
messageC = idmessage_new();
idmessage_setdescription(messageC, 'classifier');
idmessage_setfidx(messageC, 1001);
idmessage_setfamilytype(messageC, idmessage_familytype('biosig'));

messageF = idmessage_new();
idmessage_setdescription(messageF, 'feedback');
idmessage_setfidx(messageF, 999);
idmessage_setfamilytype(messageF, idmessage_familytype('biosig'));

messageD = idmessage_new();
idmessage_setdescription(messageD, 'buggymodule');
idmessage_setfidx(messageD, 990);
idmessage_setfamilytype(messageD, idmessage_familytype('biosig'));

% Imagine we receive:
% — two biosig events from the classifier (1 and 2)
% — three biosig events from the feedback (7, 8 and 9)
idmessage_setevent(messageC, 1); idasclient_add(client, messageC);
idmessage_setevent(messageC, 2); idasclient_add(client, messageC);
idmessage_setevent(messageF, 7); idasclient_add(client, messageF);
idmessage_setevent(messageD, 666); idasclient_add(client, messageD);
idmessage_setevent(messageF, 8); idasclient_add(client, messageF);
idmessage_setevent(messageF, 9); idasclient_add(client, messageF);

% At this point, our client should have in queue 5 messages, received,
% as we are imagining, within the last frame
disp(['[example_asclient] Queued messages: ' num2str(idasclient_size(client))]);

% Goodie. Now it's time to show the concept of Time—Warping in BCI design
% Imagine you want to get all the messages coming from the previous modules.
% I will also create a serializer object to print the content of the messages in
% XML format.
message = idmessage_new();
serializer = idserializerrapid_new(message);
disp('[example_asclient] Dequeing message from previous modules:');
while(true)
    found = idasclient_getprev(client, message, [], []);
    if(found == false)
        break;
    end
    buffer = idmessage_serialize(serializer);
    disp(buffer);

end

% Now we dequeue all the messages coming from the modules afterwards in the
% chain
disp('[example_asclient] Dequeing message from next modules:');
while(true)
    found = idasclient_getnext(client, message, [], []);
    if(found == false)
        break;
    end
    buffer = idmessage_serialize(serializer);
    disp(buffer);

    if(idasclient_getfidx(client) — idmessage_getfidx(message) > 1)
        disp(['  Warning: message is too delayed! ' ...
            'Module "' idmessage_getdescription(message) '" is too slow!']);
    end
end
```

5.1 XML message

The reader can refer to Section 3 for interpreting the different attributes of the XML message.

```
1 <tobiid version="0.0.2.0" description="classifier" frame="1001" ...
      family="biosig" event="1"/>
```