Blackadder Application Programming Interface

Introduction

Blackadder supports a simple API for writing Information-centric applications. Strictly speaking, there is no particular API at all regardless of Blackadder running as a user-space application or as a kernel module. The only way to communicate with Blackadder is via the Netlink socket that is opened when Blackadder starts. Therefore, an application must open a Netlink socket and send data buffers to Blackadder. These buffers must be compliant with the expected format (which one could call API) so that Blackadder interprets them as valid publish/subscribe requests. From that point of view the communication model between applications and Blackadder resembles more like a remote-procedure call model.

Blackadder responds back to applications via the Netlink socket by sending (asynchronously) data buffers that describe valid information-related events.

In this document we will describe the basic C++ wrapper for all available publish/subscribe requests that Blackadder currently expects and the events that it sends back to applications. On top of this library we have also built a number of wrappers for several languages, like Java, Python and Ruby, which will also be described.

The C++ API

The C++ API is compiled as a shared or static library and contains two singleton Classes that are available to the programmer.

The first, *Blackadder*, handles publish/subscribe requests in a blocking fashion. More specifically, all requests are sent to the networking stack before each method unblocks in the application. Note that Blackadder-related communication is asynchronous in the sense that none of the exported methods will block until an event is received.

The second Class, *NB_Blackadder*, processes publish/subscribe requests in a non-blocking fashion. A selector and a worker thread are created when the single *NB_Blackadder* object is created. That way, an application just pushes all publish/subscribe requests in a queue, which is later processed by the library.

Since both exported Classes in the library implement the singleton pattern there is no constructor available to the application. Instead, an application acquires a communication object by calling the static methods respectively for each Class:

```
static Blackadder* Instance(bool user_space);
static NB_Blackadder* Instance(bool user_space);
```

The Boolean parameter tells the library that the networking stack runs in the user or kernel space respectively. This permits the library to use the appropriate Netlink port and protocol (NETLINK_GENERIC for user space and NETLINK_BADDER for kernel space).

The description is mainly intended for understanding the behaviour of the system when publish/subscribe requests are issued by applications. For an analytic description of the expected parameters please refer to the doxygen documentation.

Publishing Scopes

A publisher can create a scope in the information structure that is maintained by the Rendezvous system by issuing a *publish_scope* request. The method that creates and sends a publish scope request is the:

```
void publish_scope(const string &id, const string &prefix_id, unsigned char
strategy, void *str_opt, unsigned int str_opt_len);
```

When a scope is published, the Rendezvous system notifies all subscribers that have previously subscribed to the scope (if such a scope exists) under which the new scope has been published about the scope publication. System notifications about information-related events are covered later in this document. Assuming that scope and information IDs are 2 bytes, represented here as hexadecimal digits, then the following method calls (called by publisher 1) would result in the information graph shown in Figure 1. Note that in the actual implementation scope and information IDs should be passed to all methods as binary arrays; *not in hex format*. Helper functions are also included for transforming from hex to binary and vice-versa (see doxygen documentation).

```
a) publish_scope("0000", "", DOMAIN_LOCAL, NULL,0)
b) publish_scope("0001", "", DOMAIN_LOCAL, NULL,0)
c) publish_scope("1111", "0000", DOMAIN_LOCAL, NULL,0)
d) publish_scope("2222", "0000", DOMAIN_LOCAL, NULL,0)
e) publish_scope("3333", "00002222", DOMAIN_LOCAL, NULL,0)
f) publish_scope("000022223333", "0001", DOMAIN_LOCAL, NULL,0)
```

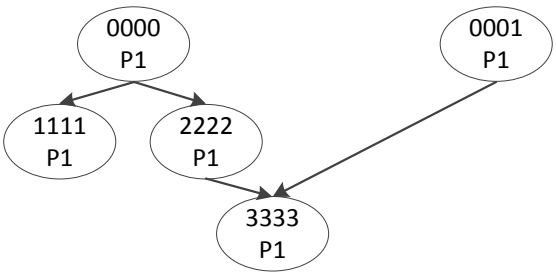

Figure 1. Information Graph - Publishing Scopes

If before call f) a subscriber was subscribed in root scope *0001* then the Rendezvous system would notify it after call f) about the republication of scope *000022223333* under scope *0001*. In this example the strategy parameter was set to DOMAIN_LOCAL, which is a strategy that defines that the visibility of the published scope is within the administrative domain of the publisher. In other words Blackadder will send this request to the rendezvous node of the domain (for that, a preconfigured LIPSIN Identifier will be used to forward the request to the RV). When publishing scopes, a publisher can specify the information to be visible only within its local host (NODE_LOCAL), its host and a

physical neighbour (LINK_LOCAL) as well as within the domain it resides (DOMAIN_LOCAL). When information visibility is set as LINK_LOCAL then Blackadder also expects a Link identifier that points to the physical neighbour. In all other cases the Link ID parameter should be NULL.

Advertising Information Items

A publisher advertises information items using the following call:

```
void publish_info(const string&id, const string&prefix_id, unsigned char strategy, void *str_opt, unsigned int str_opt_len);
```

Information items can reside under one or more scopes and never be unlinked in the information graph. Following the previous example, the following calls should result to the information structure shown in Figure 2.

a) `publish_info("0000", "", DOMAIN_LOCAL, NULL, 0)` – This is a mistake since the info is not advertised under any scope
b) `publish_info("0000", "0000", DOMAIN_LOCAL, NULL, 0)`
c) `publish_info("000A", "000022223333", DOMAIN_LOCAL, NULL, 0)`

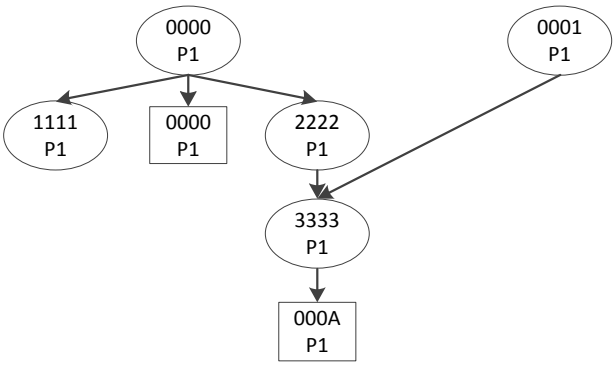

Figure 2. Information Graph - Advertising Information Items

Note that uniqueness of IDs is only enforced by the Rendezvous system within a scope only. Therefore, call b) is legal. On the contrary, call `publish_info("1111", "0000", DOMAIN_LOCAL, NULL,0)` will be rejected by the rendezvous node, since a scope with the same ID resides under scope *0000*.

Depending on the existence of subscribers in the information graph, the rendezvous node may initiate a rendezvous process during which it matches publishers and subscribers for an information item (in this case the one that is being advertised) and publishes a request for topology formation to the topology manager (if the dissemination strategy is DOMAIN_LOCAL). The Topology Manager, will then publish a notification to the publisher(s) for publishing the data for this information item. For the other supported strategies the Rendezvous node has all the necessary information to notify the publishers by itself.

Subscribing to Scopes

An end-host can subscribe to a scope by calling the following method:

```
void subscribe_scope(const string&id, const string&prefix_id, unsigned char strategy, void *str_opt, unsigned int str_opt_len);
```

The rendezvous node will create a new scope if the scope defined in the request does not exist. By subscribing to a scope, a subscriber declares to the rendezvous system that is interested in all scopes and information items that reside under that scope (and not under the whole sub-graph under that scope). Therefore, upon a scope subscription, the rendezvous node will look for any scopes or information items that are published under that scope and act appropriately.

For any scope, it will publish a request to the topology manager to create a LIPSIN Identifier and publish a notification about the existence of the underlying scope(s) to the subscriber, using the calculated Identifier.

For all information items, the rendezvous node will publish a request to the topology manager that will subsequently calculate LIPSIN identifier(s) that lead from the publisher(s) to the subscriber(s). Note that other subscribers may have previously subscribed to that scope or the specific information item and, therefore, the LIPSIN Identifier that was, then, sent to the publisher(s) must be updated. If they subscriber is interested about the whole sub-graph residing under the scope it has subscribed to, it has to manually subscribe to the descendant scopes for which it will be recursively notified by the rendezvous node upon subscription to them.

Subscribing to Information Items

An end-host can subscribe to a specific information item by calling the following method.

```
void subscribe_info(const string&id, const string&prefix_id, unsigned char strategy, void *str_opt, unsigned int str_opt_len);
```

If publishers have previously advertised this item, the rendezvous node will match them with the subscriber (and any other subscriber previously subscribed) and publish a request to the topology manager to create LIPSIN Identifiers that point from each publisher to a (potentially empty) set of subscribers and notify them about these identifiers.

In order to make things clearer, let us assume the following information structure. Publishers and subscribers that previously issued requests to the rendezvous system are also presented in the figure.

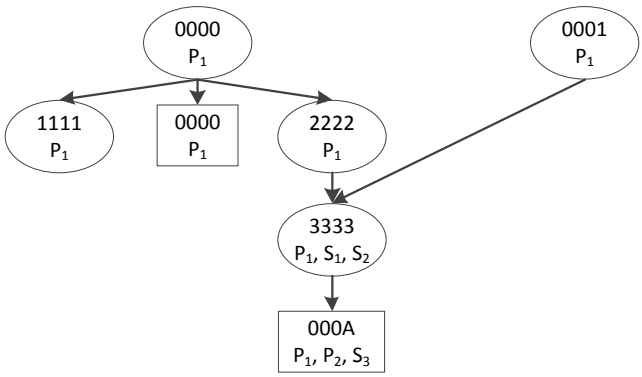

Figure 3. Information Graph - Subscribing to Information Items

Subscriber 3 then subscribes to the information item with ID *000022223333000A*. The rendezvous node will match all publishers (P_1 and P_2) with all subscribers (S_1, S_2 and S_3) and publish a topology resolution request to the topology manager. The current TM implementation finds the shortest paths from the publishers to the subscribers and constructs the respective multicast trees (along

with the respective LIPSIN IDs). Assuming that in our example, P_1 is physically the closest node to all subscribers, the topology manager will publish a notification to P_1 about the new LIPSIN ID (an update of the one sent before S_3 appeared). It will also publish a notification to P_2 notifying it about a NULL LIPSIN ID for this specific item.

A special case of scope and information item subscriptions is when the "SUBSCRIBE_LOCALLY" strategy is used. In that case no rendezvous nodes are notified. On the contrary, the subscription stays only in the local node. An example of usage of this strategy is the Topology Manager which subscribes locally to a pre-defined scope. No rendezvous needs to take place for any items under that special scope, since the only publishers are the rendezvous nodes of a domain. Rendezvous nodes a priory know a forwarding identifier that points to the node(s) on top of which the Topology Manager runs as an application.

Unpublishing Scopes

Unpublishing a scope from the information graph is accomplished by calling the following method:

```
void unpublish_scope(const string&id, const string&prefix_id, unsigned char strategy, void *str_opt, unsigned int str_opt_len);
```

This method will try to unpublish all information items that are children of that scope as described in the next subsection. If that succeeds and no other subscopes under the scope defined in the method exist, the scope is unpublished. If the scope was published under multiple scopes, only the specific branch in the graph is deleted. Otherwise the scope is deleted. All subscribers that have subscribed to the father scope(s) of the deleted scope are notified for this event.

Unpublishing Information Items

Unpublishing an information item is straightforward and is done by calling the following method.

```
void unpublish_info(const string&id, const string&prefix_id, unsigned char strategy, void *str_opt, unsigned int str_opt_len);
```

As a result the publisher issued that request will be removed by that information item in the information graph. If there are no other publishers or subscribers, the item is also be deleted by the rendezvous system. If more publishers and subscribers exist for this item, rendezvous will take place again since the LIPSIN IDs must be updated by the topology manager (e.g. in case the publisher that issued the request was close to some or all subscribers).

As described above, an information item may be advertised under multiple scopes. If this is the case when an *unpublish_info* is issued, then, assuming there are no other publishers or subscribers for the given Information ID (that defines one of the paths in the information graph), the Rendezvous node will delete the information item from the scope with *prefix_id*.

Unsubscribing from Scopes

A subscriber unsubscribes from a scope by issuing the following the request:

```
void unsubscribe_scope(const string&id, const string&prefix_id, unsigned char strategy, void *str_opt, unsigned int str_opt_len);
```

The subscriber is removed by the subscribers' list of the scope. If there are no other publishers and subscribers as well any items or subscopes the rendezvous node deletes the item from the information graph.

Unsubscribing from Information Items

A subscriber unsubscribes from an information item by issuing the following the request:

```
void unsubscribe_info(const string&id, const string&prefix_id, unsigned char strategy, void *str_opt, unsigned int str_opt_len);
```

The subscriber is removed by the information item and if there are no other publishers and subscribers, the rendezvous node deletes the item from the information graph. In the opposite case rendezvous will take place since the Forwarding identifier previously sent to one or more subscribers must be updated.

Blackadder events

Applications receive events from Blackadder as data buffers sent to their netlink sockets.

"Scope published" Events

A "scope published" (SCOPE_PUBLISHED) event is received by a subscriber whenever a new scope is created under a scope to which the subscriber has previously subscribed. Note that a subscriber will also receive such events when subscribing to a scope if sub-scopes existed before the subscription. As mentioned before, subscribing to a scope does not mean subscribing to the whole sub-graph under that scope. Therefore, subscribers receive notifications about new scopes only when these are published directly under the scope for which the subscription was issued. The "scope published" event is accompanied by the Information ID of the new scope. Scopes and information items may be reachable by multiple paths in the information graph and, thus, identified by multiple information IDs. However, events sent from Blackadder contain only a single information ID; the one that is relevant with the publish/subscribe request previously issued by an application.

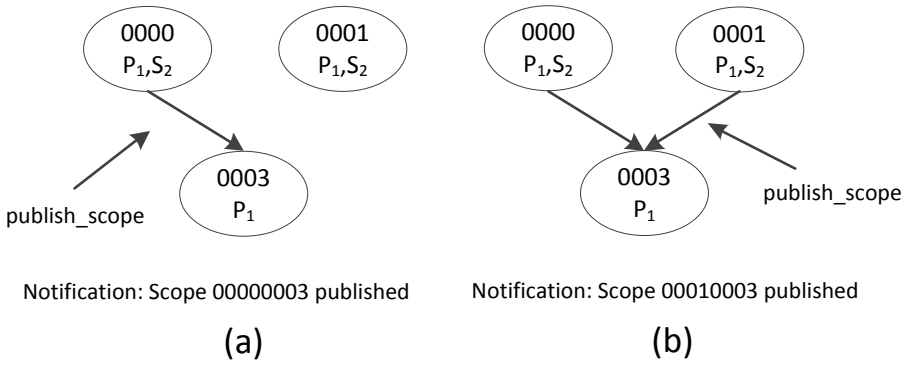

Figure 4. Receiving notifications about new scopes

As we observe in the Figure above, Subscriber S_2 is subscribed to both root scopes in the graph. Then, Publisher P_1 publishes a new scope under scope *0000* (figure 3(a)), an action that triggers a notification to S_1. The notification contains the Information ID *00000003*. When P_1 republishes the scope under root scope *0001* (figure 3(b)), another notification is sent to S_1, containing the information ID *00010003*. End-nodes along with the applications running on them do not know the whole information graph. They may only know sub-parts of it if they have previously issued

publish/subscribe requests for these parts. Therefore, in the previous example there is no way for a node to say if these two notifications refer to the same scope. Only Rendezvous nodes that administer the information (e.g. the rendezvous node of a domain) have a global knowledge about the information structure.

"Scope unpublished" Event

When a scope is unpublished and removed from the information graph, subscribers of the father scope(s) are notified about that event (SCOPE_UNPUBLISHED). As with new scopes, only subscribers to scopes residing one level higher than the deleted scope are notified.

Publishing Data

Applications publish data for specific Information IDs by calling the following method:

```
void publish_data(const string&id, unsigned char strategy, void *str_opt,
unsigned int str_opt_len, void *data, unsigned int data_len);
```

However, publications are never sent to the network or to other local subscribers unless some kind of Rendezvous has been previously took place. When an explicit Rendezvous takes place in a Rendezvous node, Blackadder receives Forwarding Identifiers by the topology manager and assigns them to an information item for which local publishers exist. If a publisher tries to publish data before such an assignment, the request will be rejected by Blackadder. On the opposite case, Blackadder will publish the data to the network (and to local subscribers) using the assigned Forwarding Identifier.

There also exist cases where no explicit rendezvous takes place. In these cases a publisher assumes that for a specific information item one or more subscribers exist and, either provides a Forwarding identifier to Blackadder or instructs Blackadder to reuse another Forwarding Identifier. This is achieved by using the "IMPLICIT_RENDEZVOUS" strategy and providing an FID. If the Forwarding Identifier is not an "all-zero" identifier, Blackadder will publish the data to the network regardless of the existence of subscribers. If FID is an "all-zero" identifier, Blackadder will use the FID (if one is assigned by the rendezvous system) of the item (if such an item exists) that is one level higher in the information graph.

An example of the former case is how the Topology Manager publishes notifications to publishers and subscribers. The TM is able to calculate such FIDs, so it uses them along with a "IMPLICIT_RENDEZVOUS" strategy to reach network nodes.

The latter case could be used for fragmenting an information item for which rendezvous has previously taken place. In such a case, an FID is assigned to an information item. A publisher, then, publishes information items that reside under that item by requesting a "IMPLICIT_RENDEZVOUS" strategy with an "all-zero" FID. Blackadder will publish these items using the forwarding identifier assigned to the "father" item.

Below we describe the two events that a publisher should expect after advertising an information item using one of the supported strategies.

"Start Publish" Event

A "Start Publish" (START_PUBLISH) event is sent by Blackadder to an application whenever an FID is assigned to an advertised information item. Note that a publisher does not receive any event when the FID is updated because subscribers for that item leave or join, although Blackadder internally updates the assigned FID. Therefore, a publisher is notified only once when subscribers first join.

As with the previously described events, a "Start Publish" event contains the Information ID identifying the information Item to be published. In the example presented in the following figure, publishers P_1 and P_2 have already advertised the information item *0000000A* (a). Then, P_2 republishes the same item under scope *0001* (b). The publishers' set for each path in the graph is shown in brackets. Finally, Subscriber S_2 subscribes to the root scope *0001 (c)*. At that point rendezvous will happen in the rendezvous node that administers the information structure (according to the strategy).

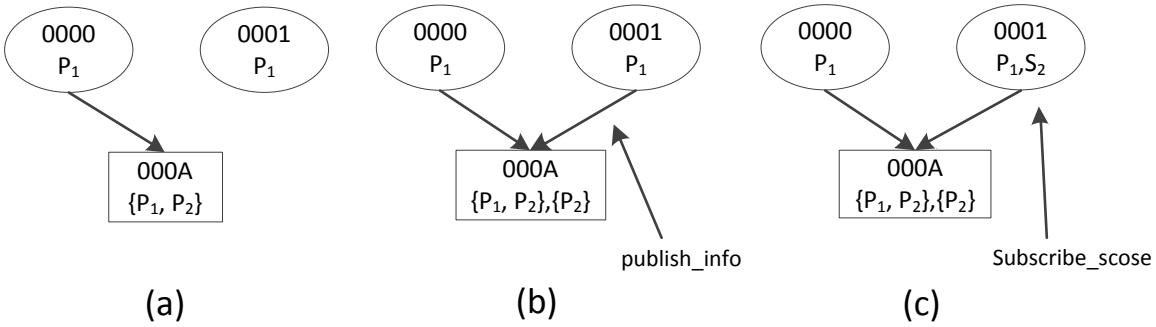

Figure 4. "Start Publish" Events

Let us now assume that P_1 is the closest node to S_2 and, therefore, it is the one that receives a notification from the Topology Manager about an FID. Blackadder running in that node assigns the information item *0000000A* with an FID and sends to the publisher a "Start Publish" event. Apart from that, Blackadder also stores the alternative ID (*0001000A*), which was included in the TM's publication, in the information item. Note that the event sent to the application will include only the ID *0000000A*, since the publisher does not even know about the alternative one. Respectively, when data is published to the network all known Information IDs are included in the publication. Each Blackadder node that receives the data sends to the subscribers the ID that is relevant with the pending subscriptions.

"Stop Publish" Event

A "Stop Publish" (STOP_PUBLISH) event is sent by Blackadder to an application whenever no more subscribers exist for an information item, which has been previously assigned with an FID; that is when subscribers existed. Note that the semantics of both "Start" and "Stop" events should be translated by the application. For instance, a video streamer would constantly publish different video chunks using the same Information ID (like a channel) upon the reception of a "Start Publish" event and would stop publishing upon the reception of a "Stop Publish" event. On the other hand a photo publisher would publish only once after receiving a "Start Publish" event assuming that all subscribers would unsubscribe after receiving the data.

"Published Data" Event

Finally, a "Published Data" (PUBLISHED_DATA) event is sent to an application (subscriber) when data has arrived for a specific publication. This event contains the Information ID as well as the received data.

"Undefined" Event

If an error occurs or the `getEvent()` operation (see below) is interrupted, an event of type UNDEF_EVENT (0) is returned.

Event Handling

The two singleton Classes provide a different way of handling events sent by the networking stack. Using the blocking version, an application must call the method:

```
void getEvent(Event &ev);
```

This method is blocking and, thus, an application will block until an Event is sent by Blackadder. Note that after the call, the event's internal buffer is freed when the event is destroyed, but not when the same event is reused in a new *getEvent* call (in the new call, a new buffer is still allocated).

The non-blocking version provides a callback mechanism for receiving network events. More specifically, an application should first a callback method that will be called by the library when Events are received by Blackadder:

```
void setCallback(callbacktype t);
```

The callback method should be of type `void callbacktype(Event *);` and must process the event and free the memory allocated for the Event by the library.

SWIG-based Language Bindings

We provide language bindings also for Python, Ruby and Java. These bindings are generated with SWIG (Simplified Wrapper and Interface Generator), but no knowledge of SWIG itself is required in order to compile, install and use these bindings.

The SWIG-based language bindings wrap (most) of the functions in the C++ API so that that they can be used in "higher level" languages. The bindings can be compiled and installed at the same time with the C++ library. The semantics of the SWIG-based APIs are also the same as described above for the C++ API. Thus general knowledge of the C++ API is a prerequisite for reading this section.

Python API

Importing

The SWIG-based Python API is installed in a package called *blackadder* that contains a module that is also called *blackadder*. This module can be imported the following way:

```
from blackadder import blackadder
```

Alternatively, all classes, functions and constants can be imported from the module like this:

```
from blackadder.blackadder import *
```

In order to examine what is in the module, type (e.g.) `dir(blackadder)` or `dir()` and `help(blackadder)` or `help()` in a Python interpreter.

Instantiation

A new Blackadder instance (blocking or non-blocking) is created with the `Blackadder.Instance(user_space)` or `NB_Blackadder.Instance(user_space)` functions, e.g.:

```
ba = Blackadder.Instance(True)
```

The Blackadder IPC connection can be disconnected with the `disconnect()` method.

Pub/Sub Methods

The API functions (i.e., methods of a Blackadder instance) correspond to the ones found in the C++ API. *ID*s, *strategy options* and *data* parameters are given as (binary) string or buffer objects, while *strategies* are given as integers. Notably, both *str_opt_len and data_len* paramers are omitted as the length information is implicitly maintained in the corresponding string/buffer objects. Hence, the methods of the basic pub/sub API are:

```
publish_scope(id, prefix_id, strategy, str_opt)
publish_info(id, prefix_id, strategy, str_opt)
unpublish_scope(id, prefix_id, strategy, str_opt)
unpublish_info(id, prefix_id, strategy, str_opt)

subscribe_scope(id, prefix_id, strategy, str_opt)
subscribe_info(id, prefix_id, strategy, str_opt)
unsubscribe_scope(id, prefix_id, strategy, str_opt)
unsubscribe_info(id, prefix_id, strategy, str_opt)

publish_data(id, strategy, str_opt, data)
```

Usage example:

```
prefix_id = '\x0A\x00\x00\x00\x00\x00\x00\x0B'
id = '\x0C\x00\x00\x00\x00\x00\x00\x0D'

ba.publish_scope(prefix_id, '', NODE_LOCAL, None)
ba.publish_info(id, prefix_id, NODE_LOCAL, None)

data = '123ABC'
ba.publish_data(prefix_id + id, NODE_LOCAL, None, buffer(data, 0, 3))
```

Note that in the non-blocking API, the *data* buffer given to the `publish_data()` function have to be allocated either with `new_malloc_buffer(data_len)` or with `to_malloc_buffer(data)`.

```
nb_ba.publish_data(prefix_id + id, NODE_LOCAL, None,
                   to_malloc_buffer(data[0:3]))
```

Events

Event objects are also included in the API. A new event can be created by calling `Event()` and acquired from Blackadder by the `getEvent(ev)` method as in the blocking C++ API. The accessors are also the same as in the C++ API: *type* (int), *id* (buffer), *data* (buffer), and *data_len* (int). The length of *data* is implicitly set to *data_len*. Usage example:

```
ev = Event()
ba.getEvent(ev)
print '%d%r%r' % (ev.type, ev.id, ev.data)
```

When using the non-blocking API, events are handled in an event handler (callback) that is defined as a single-argument function and "decorated" with `@blackadder_event_handler` (for correct handling of Event objects) like this, for example:

```
@blackadder_event_handler
def event_handler(ev):
    print '%d%r%r' % (ev.type, ev.id, ev.data)
```

The event handler function is assigned with the `setPyCallback(event_handler)` method, e.g.:

```
nb_ba.setPyCallback(event_handler)
```

The `join()` method from the C++ API is also available in Python:

```
nb_ba.join()
```

Ruby API

Loading

The experimental SWIG-based Ruby API library is called *blackadder* and is loaded like this:

```
require 'blackadder'
```

To see what's in the library, you can run, e.g., `Blackadder.constants.sort` and `Blackadder::Blackadder.methods.sort` in the *irb* command line interpreter.

Instantiation

A Blackadder instance (blocking or non-blocking) is created with the `Blackadder::Instance(boolean user_space)` or `NB_Blackadder::Instance(boolean user_space)` methods as shown below.

```
ba = Blackadder::Blackadder::Instance(true)
```

The Blackadder IPC connection can be disconnected with the `disconnect()` method.

Pub/Sub Methods

The pub/sub methods in the Ruby API are the same as in the C++ and Python APIs. *ID*s, *strategy options* and *data* are given as strings. *Strategies* are given as numbers.

```
publish_scope(id, prefix_id, strategy, str_opt)
publish_info(id, prefix_id, strategy, str_opt)
unpublish_scope(id, prefix_id, strategy, str_opt)
unpublish_info(id, prefix_id, strategy, str_opt)

subscribe_scope(id, prefix_id, strategy, str_opt)
subscribe_info(id, prefix_id, strategy, str_opt)
unsubscribe_scope(id, prefix_id, strategy, str_opt)
unsubscribe_info(id, prefix_id, strategy, str_opt)

publish_data(id, strategy, str_opt, data)
```

Usage example:

```
prefix_id = 0xA.chr + 0x0.chr*6 + 0xB.chr
id = 0xC.chr + 0x0.chr*6 + 0xD.chr

ba.publish_scope(prefix_id, '', Blackadder::NODE_LOCAL, nil)
ba.publish_info(id, prefix_id, Blackadder::NODE_LOCAL, nil)

data = '123ABC'
ba.publish_data(prefix_id + id, Blackadder::NODE_LOCAL, nil, data[0, 3])
```

Note that in the non-blocking API, the *data* buffer given to the `publish_data()` function have to be re-allocated with `to_malloc_buffer(data)`. For example:

```
nb_ba.publish_data(prefix_id + id, Blackadder::NODE_LOCAL, nil,
                   to_malloc_buffer(data[0, 3]))
```

Events

Event objects can be created with `Event::new` and acquired with the `getEvent(ev)` method in the blocking API, or by defining and setting (with the `setRubyCallback(object, method)` method) an event handler in the non-blocking API. Note that these functions might cause problems with multi-threaded Ruby code.

Blocking example:

```
ev = Blackadder::Event::new
ba.getEvent(ev)
p ev
p ev.type
p ev.data  # length implicitly set to data_len
```

Non-blocking example:

```
class EventHandler
    def event_handler(ev)
        p ev
        p ev.type
    end
end
evh = EventHandler.new
nb_ba.setRubyCallback(evh, 'event_handler')
```

The `join()` method from the C++ API is also available in Python:

```
nb_ba.join()
```

Java API (SWIG)

Loading

The experimental SWIG-based Java API library is called *blackadder_java* and is loaded like this:

```
static {
    System.loadLibrary("blackadder_java");
}
```

For convenience, the *blackadder_java.BA* class, which contains Blackadder constants etc., can be "renamed" to simply *BA* like this:

```
static final blackadder_java.BA BA = null;
```

Instantiation

A Blackadder instance (blocking) is created with the `Blackadder.Instance(user_space)` method as shown below. The non-blocking version of the library is currently not supported in Java.

```
blackadder_java.Blackadder ba = blackadder_java.Blackadder.Instance(true);
```

The Blackadder IPC connection can be disconnected with the `disconnect()` method.

Pub/Sub Methods

The pub/sub methods in the SWIG-based Java API are the same as in the C++ and Python APIs. *ID*s, *strategy options* and *data* are given as byte arrays. *Strategies* are given as integers.

```
publish_scope(byte[] id, byte[] prefix_id, int strategy, byte[] data)
publish_info(byte[] id, byte[] prefix_id, int strategy, byte[] data)
unpublish_scope(byte[] id, byte[] prefix_id, int strategy, byte[] data)
unpublish_info(byte[] id, byte[] prefix_id, int strategy, byte[] data)

subscribe_scope(byte[] id, byte[] prefix_id, int strategy, byte[] data)
subscribe_info(byte[] id, byte[] prefix_id, int strategy, byte[] data)
unsubscribe_scope(byte[] id, byte[] prefix_id, int strategy, byte[] data)
unsubscribe_info(byte[] id, byte[] prefix_id, int strategy, byte[] data)

publish_data(byte[] id, int strategy, byte[] str_opt, byte[] data)
```

Usage example:

```
prefix_id = { 0x0A, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0B };
id = { 0x0C, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0D };
full_id = [...] // use e.g. System.arraycopy() to get prefix_id + id

ba.publish_scope(prefix_id, new byte[0], BA.NODE_LOCAL, null);
ba.publish_info(id, prefix_id, BA.NODE_LOCAL, null);

byte[] data = { '1', '2', '3', 'A', 'B', 'C' };
ba.publish_data(full_id, BA.NODE_LOCAL, null, data);
```

Events

Blackadder Events are objects in Java. They can be acquired with the blocking `getEvent(ev)` method, and their content is accessed via accessor methods such as `getType()`, `getId()`, `getData_len()`, and `getData()`.

Example:

```
blackadder_java.Event ev = new blackadder_java.Event();
ba.getEvent(ev);
int type = ev.getType();
byte[] id = ev.getId();
long data_len = ev.getData_len();
byte[] data = ev.getData();
```

Independent C Wrapper

The independent, experimental C wrapper makes it possible to use the Blackadder library also from C programs.

Including

The Blackadder C library can be included into a C program like this:

```
#include <blackadder_c.h>
```

Instantiation

Use the `ba_instance(int user_space)` or `nb_ba_instance(int user_space)` functions (blocking or non-blocking) that return a `ba_handle` or `nb_ba_handle` struct respectively. E.g.:

```
ba_handle ba = ba_instance(1);
```

The Blackadder IPC connection can be disconnected with the `ba_isconnect(ba_handle ba)` function and deleted with the `ba_delete(ba_handle ba)` function.

Pub/Sub Methods

The pub/sub methods are the same as in the C++ API. The names of the functions start with *ba_* and the first parameter is a handle to the Blackadder instance. *ID*s are given as `(char *id, unsigned int id_len)` pairs instead of strings. Two examples below; as usual, all (un)publish / (un)subscribe scope / info functions take the same set of parameters.

```
void ba_publish_info(ba_handle ba, const char *id, unsigned int id_len,
                     const char *prefix_id, unsigned int prefix_id_len,
                     unsigned char strategy,
                     void *str_opt, unsigned int str_opt_len);

void ba_publish_data(ba_handle ba, const char *id, unsigned int id_len,
                     unsigned char strategy,
                     void *str_opt, unsigned int str_opt_len,
                     void *data, unsigned int data_len);
```

Usage example:

```
const char prefix_id[] = {0x0A, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0B};
const char id[] = {0x0C, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x0D};
char full_id[] = [...]; // use e.g. memcpy() to get prefix_id + id

ba_publish_scope(ba, prefix_id, PURSUIT_ID_LEN, "", 0,
                 NODE_LOCAL, (void *)0, 0);
ba_publish_info(ba, id, PURSUIT_ID_LEN, prefix_id, PURSUIT_ID_LEN,
                NODE_LOCAL, (void *)0, 0);

char data[] = "123ABC";
ba_publish_data(ba, full_id, PURSUIT_ID_LEN, NODE_LOCAL, (void *)0, 0,
                data, 3);
```

Events

Event "objects" can be created with the `ba_event_new()` function that returns an event handle (`ba_event`). Correspondingly, that handle can be deleted with `ba_event_delete(ba_event`

`ev)`. Event's can be acquired from Blackadder either via the blocking `ba_get_event(ba_handle ba, ba_event ev)` function or, if the non-blocking version of the library is used, an event handler function that is assigned with `nb_ba_set_callback(nb_ba_handle nb_ba, nb_ba_callbacktype cf)`. Event fields are accessed via accessor functions:

```
void ba_event_type(ba_event ev, unsigned char **type);
void ba_event_id(ba_event ev, const char **id, unsigned int *id_len);
void ba_event_data(ba_event ev, void **data, unsigned int **data_len);
```

Blocking example:

```
ba_event ev = ba_event_new();
unsigned char *_type = NULL, type;
ba_get_event(ba, ev);
ba_event_type(ev, &_type);
type = *type;
ba_event_delete(ev);
```

Non-blocking example:

```
static void event_handler(ba_event ev) {
    unsigned char *_type = NULL, type;
    ba_event_type(ev, &_type);
    type = *type;
    // ...
    ba_event_delete(ev);
}

nb_ba_set_callback(ba, event_handler);
```

The `join()` method from the C++ API is also available in C:

```
nb_ba_join(nb_ba)
```

Independent Java Wrapper

The independent Java Wrapper is a non-SWIG Java binding for Blackadder. This particular component offers a slight improvement in performance due to reduced buffer copies between the Blackadder library and JVM when receiving data packets, as well as a few classes that provide an object-oriented abstraction to the network functionalities. The component is located in *{blackadder_dir}lib/bindingsjava-binding*. To compile and build the binding, follow the instructions in the HowTo document.

Other than the Java interface, this component offers no extra functionality to Blackadder. The semantic of each operation is the same as in Blackadder, therefore the developer should first consult with the operations provided by the native C++ API and then proceed with the Java wrapper.

Directory Structure

The main *java-binding* directory is an Eclipse IDE project folder. If you are using Eclipse, you can easily import *BlackadderJava* to your workspace by choosing

"File -> Import ->Existing Project into workspace"

If you don't use Eclipse, an Ant build file is also provided (in this case, Apache Ant must be installed on your system).

The structure of the main directory is as follows:

- The *src* directory contains the source files.
- The *lib* directory contains library files used. Currently, only Apache Common Codec is used.
- .classpath and .project used by Eclipse.
- build.xml is the ant build file.
- the bin directory contains the built classes
- the dist directory contains the created jar file (*ant dist*)

Usage

Core classes

The `eu.pursuit.core` package contains the classes that represent the basic entities of the system, such as identifiers and scopes. The class `ByteIdentifier` represents an identifier used in information item names, either Rendezvous or Scope identifiers. The class is actually a wrapper of a byte array, i.e. it contains a single instance field

```
private final byte[] id;
```

The constructor `ByteIdentifier(byte[] id)` takes as a parameter an existing byte array and wraps it (it just copies the reference; later changes to the array's content mutates the object as well). The `getId()` method returns a reference to the byte array. For convenience, the constructor `ByteIdentifier(byte value, int length)` initializes the instance with a `length` size array whose every item equals `value`.

Rendezvous identifiers (RIDs) are be represented by `ByteIdentifier` instances. You need to compute the byte array first yourself (e.g. by applying a hash function) and then wrap it in a `ByteIdentifier` instance. Blackadder will not transform human readable strings to binary format. This is task must be performed by the application.

The class `ScopeID` represents scope identifiers. Scopes in Blackadder have the form of *Scope1/Scope2/Scope3* and so on, with Scope1, Scope2 and Scope3 being simple identifiers. ScopeID objects are internally organized as a list of ByteIdentifiers

```
private final List<ByteIdentifier> list;
```

The class provides a constructor that takes as input an existing list of ByteIdentifier instances and copies the list contents to its own list (it copies the references of the items in the list) . The default constructor creates an instance with an empty list. You can also create new empty scope identifiers through the static factory method `createEmptyScope()`. The method `addSegment(ByteIdentifier)` adds the identifier passed as input to the end of the list. The method returns a reference to `this` so it can be sequentially called. For example

```
ByteIdentifier scope1, scope2, scope3;
//scope1, scope2 and scope3 are initialized
```

```
ScopeID scopeID = ScopeID.createEmptyScope();
scopeID.addSegment(scope1).addSegment(scope2).addSegment(scope3);
```

By default, each identifier (Rendezvous or Scope segment) is a 64bit value, or 8 bytes. This value is also kept also in the class' static field `SEGMENT_SIZE`. If you want to change the value of segment size, change this field accordingly; the field is public and not final.

Class `ItemName` in the same package represents an item name, which is the pair <SID, RID>. In Java, an `ItemName` instance contains a `ScopeID` (SID) and a `ByteIdentifier` (RID).

Publishing and subscribing to information

To interact with the network, you first have to obtain an instance of a `BlackAdderClient` object. The class is located in the `eu.pursuit.client` package. To get the instance, you must first set up the component by calling

```
BlackAdderWrapper.configureObjectFile(String path_to_so)
```

giving as parameter the location of the object file created during the compilation of the java-binding (build instructions are in the HowTo document). Once you do that, you may get the `BlackAdderClient` instance by invoking

```
BlackAdderClient.getInstance();
```

The `BlackAdderClient` instance provides all the methods for publishing and substring to information, incuding:

- `publishScope()`
- `publishInfo()`
- `unpublishScope()`
- `unpublishInfo()`
- `subscribeScope()`
- `subscribeInfo()`
- `unsubscribScope()`
- `unsubscribeInfo()`

In these methods you need to specify the name of the published/requested information in the form of `ScopeID`, `ByteIdentifier` (for RIDs) and `ItemName` instances. These methods also require specifying the dissemination strategy used as well as dissemination strategy options. Dissemination strategies are represented by the `Strategy` enum, located in the `eu.pursuit.core` package. Currently, the enum defines the following strategies

- NODE
- LINK_LOCAL
- DOMAIN_LOCAL
- IMPLICIT_RENDEZVOUS
- BROADCAST_IF

Strategy options is an open field for experimentation. To pass strategy options to Blackadder, create your own classes that implement the `StrategyOptions` interface. The interface provides a few methods for serializing/desterilizing the options to/from binary format. In case the Strategy used is

IMPLICIT_RENDEZVOUS, the options passed should be the FID pointing to the recipient node. FIDs are represented by the `ForwardIdentifier` class, also in the the `eu.pursuit.core` package.

To transmit actual data to the network, use the `publishData()`method from the `BlackAdderClient` instance. There are several overloads of this method.

The following code snippet shows an example of how to publish data named Sid1/Sid2/Rid using the DOMAIN_LOCAL dissemination strategy.

```
ScopeID scope = ScopeID.createEmptyScope();
ByteIdentifier sid1, sid2, rid;

//identifiers initialized

scope.addSegment(sid1).addSegment(sid2);
ItemName name = new ItemName(scope, rid);
byte [] data;

//data initialized
. . .

Publication pub = new Publication(name, data);
client.publishData(pub, Strategy.DOMAIN_LOCAL);
```

Receiving notifications from the network

Messages from the network are captured using the `getNextEvent()`of the `BlackAdderClient` instance. For example,

```
Event ev = BlackAdderClient .getNextEvent();
```

Similar to the C++ API, this method will block until the next message from the network arrives. The `Event` class –also `in the` `eu.pursuit.core` package- represents incoming notifications. The `getType()` method returns the type of the event represented by an `EventType` enum (START_PUBLISH, STOP_PUBLISH, SCOPE_PUBLISHED, DATA). To get the id of the item for which this notification arrived use the `getId()` method. This method returns the item name in byte [] format. If the notification regards an information item (that means not a scope), you may easily transform it to an `ItemName` instance via

```
ItemName.parseSerializedName(byte[] array, int segmentSize)
```

where `array` is the id contained in the event and `segmentSize` is the size of each segment. The method splits the id in x pieces of `segmentSize`, assigns the x-1 first segments to a `ScopeId` instance and the last piece to a `ByteIdentifier` (RID). If the event contains data, you may get a copy of the buffer via `getDataCopy()` or a certain byte value via `getByte(int)`.

IMPORTANT: In the case of data events, always remember to call `freeNativeBuffer()` before disposing the object, in order to free the underlying native c buffer.

Closing the application

When you application is about to terminate, remember to properly close the connection with Blackadder. Call `the disconnect()` method from your `BlackAdderClient` instance.