Blackadder Application Programming Interface

# Introduction

Blackadder supports a simple API for writing Information-centric applications. In fact there is no 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 so called 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 data buffers that describe valid information-related events. In this document we will describe the C/C++ wrapper for all available publish/requests that Blackadder currently expects and the events that it sends back to applications.

# The C/C++ API

The C/C++ API is implemented in the blackadder.cpp/hpp files which define a single class called Blackadder. The constructor of the class takes a Boolean argument that defines whether Blackadder runs as a user space process or as a kernel module. Actually, this only influences only the instantiation of the application’s Netlink socket.

## 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:

*publish\_scope(string &id, string &prefix\_id, char strategy, char \*LID)*

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.

1. *publish\_scope(“0000”, “”, DOMAIN\_LOCAL, NULL)*
2. *publish\_scope(“0001”, “”, DOMAIN\_LOCAL, NULL)*
3. *publish\_scope(“1111”, “0000”, DOMAIN\_LOCAL, NULL)*
4. *publish\_scope(“2222”, “0000”, DOMAIN\_LOCAL, NULL)*
5. *publish\_scope(“3333”, “00002222”, DOMAIN\_LOCAL, NULL)*
6. *publish\_scope(“000022223333”, “0001”, DOMAIN\_LOCAL, NULL)*



Figure . 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:

publish\_info(string &id, string &prefix\_id, char strategy, char \*LID)

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.

1. *publish\_info(“0000”, “”, DOMAIN\_LOCAL, NULL) – This is a mistake since the info is not advertised under any scope*
2. *publish\_info(“0000”, “0000”, DOMAIN\_LOCAL, NULL)*
3. *publish\_info(“000A”, “000022223333”, DOMAIN\_LOCAL, NULL)*



Figure . 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)* 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:

*subscribe\_scope(string &id, string &prefix\_id, char strategy, char \*LID)*

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.

*subscribe\_info(string &id, string &prefix\_id, char strategy, char \*LID)*

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 . Information Graph - Subscribing to Information Items

Subscriber 3 then subscribes to the information item with ID *000022223333000A.* The rendezvous node will match all publishers (P1 and P2) with all subscribers (S1, S2 and S3) 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, P1­ is physically the closest node to all subscribers, the topology manager will publish a notification to P1 about the new LIPSIN ID (an update of the one sent before S3 appeared). It will also publish a notification to P2 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:

*unpublish\_scope(string &id, string &prefix\_id, char strategy, char \*LID)*

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.

*unpublish\_info(string &id, string &prefix\_id, char strategy, char \*LID)*

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:

*unsubscribe\_scope(string &id, string &prefix\_id, char strategy, char \*LID)*

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:

*unsubscribe\_info(string &id, string &prefix\_id, char strategy, char \*LID)*

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.

### “New Scope” Events

A “new scope” 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 “new scope” 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 S2 is subscribed to both root scopes in the graph. Then, Publisher P1 publishes a new scope under scope *0000* (figure 3(a)), an action that triggers a notification to S1. The notification contains the Information ID *00000003*. When P1 republishes the scope under root scope *0001* (figure 3(b)), another notification is sent to S1, 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.

### “Deleted Scope” Event (not implemented yet)

When a scope is unpublished and removed from the information graph, subscribers of the father scope(s) are notified about that event. 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:

*publish\_data(string &id, char strategy, char \* FID, char \*data, 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 “PUBLISH\_NOW” 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 “PUBLISH\_NOW” 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 “PUBLISH\_NOW” 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” 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 P1 and P2 have already advertised the information item *0000000A* (a). Then, P2 republishes the same item under scope *0001* (b). The publishers’ set for each path in the graph is shown in brackets. Finally, Subscriber S2 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 . "Start Publish" Events

Let us now assume that P1 is the closest node to S2 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” 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” 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.