BlackadderJava is a thin layer that provides application developers a Java API on top the Blackadder prototype.

1. Requirements

To compile and run BlackadderJava you need Java 1.5+ installed and the blackadder click module installed and running at your host. The current version of BlackadderJava is compatible with blackadder v.0.1.

2. Directory Structure

The main BlackadderJava 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 to compile and build BlackaddJava (in this case, apache ant must be installed on your system). After you download the BlackadderJava, the structure of the main directory is as follows:

- The *src* directory contains the source files.
- The *lib* directory contains library files used by BlackadderJava. Currently, only Apache Common Codec is used.
- .classpath and .project used by Eclipse.
- build.xml is the ant build file.

Compiled classes are placed in a bin directory, which is created during the build process (either by eclipse or the ant script).

3. BlackadderJava Usage

3.1. Core classes

Package `eu.pursuit.core` contains the classes that represent the basic entities of the system, such as publications, 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 copies the reference, meaning that later changes to the array's content mutates the object as well). The `getId()` method returns a reference to the byte array (again, changing the contents of the array will mutate the `ByteIdentifier` instance). For testing convenience, the constructor `ByteIdentifier(byte value, int length)` initializes the instance with a `length` size array whose every item equals `value`.

Rendezvous identifiers 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.

Scopes have the form of *Scope1/Scope2/Scope3* and so on, with Scope1, Scope2 and Scope3 being simple identifiers. The class `ScopeID` represents scope identifiers, which 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 (that is, 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);
```

In blackadder v0.1., each identifier (Rendezvous or Scope segment) is a 64bit value, or 8 bytes, kept also in the class's static field `SEGMENT_SIZE` (not that the field is not final). Constructor `ScopeID(byte[] scopeData)` takes the array passed as parameter, brakes in pieces of `SEGMENT_SIZE`, creates an identifier for each piece and adds it to the list.

When you publish or subscribe to information (see section 3.3) , you must also provide the dissemination strategy used. Dissemination strategies are represented by the `Strategy` enum, also defined in `eu.pursuit.core`. Currently, `Strategy` has the following values

- NODE
- LINK_LOCAL
- DOMAIN_LOCAL

3.2. Connecting to the blackadder module

To connect and communicate with the blackadder click module, you need to create instances of the `BlackAdderClient` class , located in package `eu.pursuit.client`, using one of the static factory `create()` methods:

```
BlackAdderClient client = BlackAdderClient.create();
```

In v0.1, `BlackAdderClient` objects connect to the blackadder module using TCP sockets. By default, the blackadder module awaits for connections at port 5000. If this is not

the case, there are various forms of the `create()` method that take as input the port (and even the host ip address) to be used. `BlackAdderClient` objects maintain the TCP connection with the click module and separate threads for reading from and writing to the socket stream.

3.3. Publishing and subscribing

To publish or subscribe to information, `BlackAdderClient` instances provide the following methods:

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

in which you must provide the scope (and perhaps the parent scope) represented as `ScopeID` instances and the dissemination strategy used. Supplementary, you may need to define whether the rendezvous if local or not (via a 0/1 byte flag) and the a forwarding identifier represented by the `eu.pursuit.core.ForwardingIdentifier` class.

To transmit actual data to the network, `BlackAdderClient` objects provide the following method

```
publishData(Publication publication, Strategy strategy);
```

where `strategy` is the dissemination strategy to be used and `Publication` is a class representing actual data. `Publication` instances have two instance fields:

1. the item's full name represented by a ScopeID object whose last segment is the rendezvous identifier
2. the data carried, represented by a byte array.

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

```
ScopeID name = ScopeID.createEmptyScope();

ByteIdentifier sid1, sid2, rid;

//identifiers initialized

name.addSegment(sid1).addSegment(sid2).addSegment(rid);

byte [] data;

//data initialized

Publication pub = new Publication(name, data);

client.publishData(pub, Strategy.DOMAIN_LOCAL);
```

3.4. Receiving notifications from the network

BlackadderJava receives messages from the click module either when notifications are sent from the network (e.g. when rendezvous takes place) or when requested data arrives. Incoming messages are represented by the `IncomingIPCMessage` class located in package `eu.pursuit.client.messages`. `BlackAdderClient` objects maintain an internal thread that receives incoming notifications and stores them in a FIFO queue. To get a incoming message, use the `nextIncoming()` method of `BlackAdderClient` objects. The `method` gets (and removes) the next message from the FIFO queue. If the queue is empty, the method blocks until a message arrives. The `nextIncoming(long timeout, TimeUnit unit)` awaits at most `timeout units` and returns `null` if nothing arrived.

After you receive an `IncomingMessage` you need identify why it was sent. Use the `getType()` method which should one of the `byte` constants defined in `IncomingIPCMessage`

1. START_PUBLISH
2. STOP_PUBLISH
3. SCOPE_PUBLISHED
4. DATA

Incoming messages also contain of the information item name that the message is related to. For example if you receive a START_PUBLISH (or STOP_PUBLISH), the message also contains the full name (Scope and Rendezvous identifier) of the item that you are requested to start (or stop) transmitting. To get the name of the item, call the `getName()` method, which returns a ScopeID instance. If the type returned is DATA, the message contains data. In this case, cast the instance to a `DataIncomingMessage` (a subclass of `IncomingMessage`) and use the `getData()` the get the byte array contained.

3.5. Example

A full example of subscribing and receiving data from the network can be found in the `VideoReceiver` class, located in package `eu.pursuit.application`.