eBus Programmer's Manual

Version: 4.4.0

Released: August 12, 2017

Welcome to eBus!

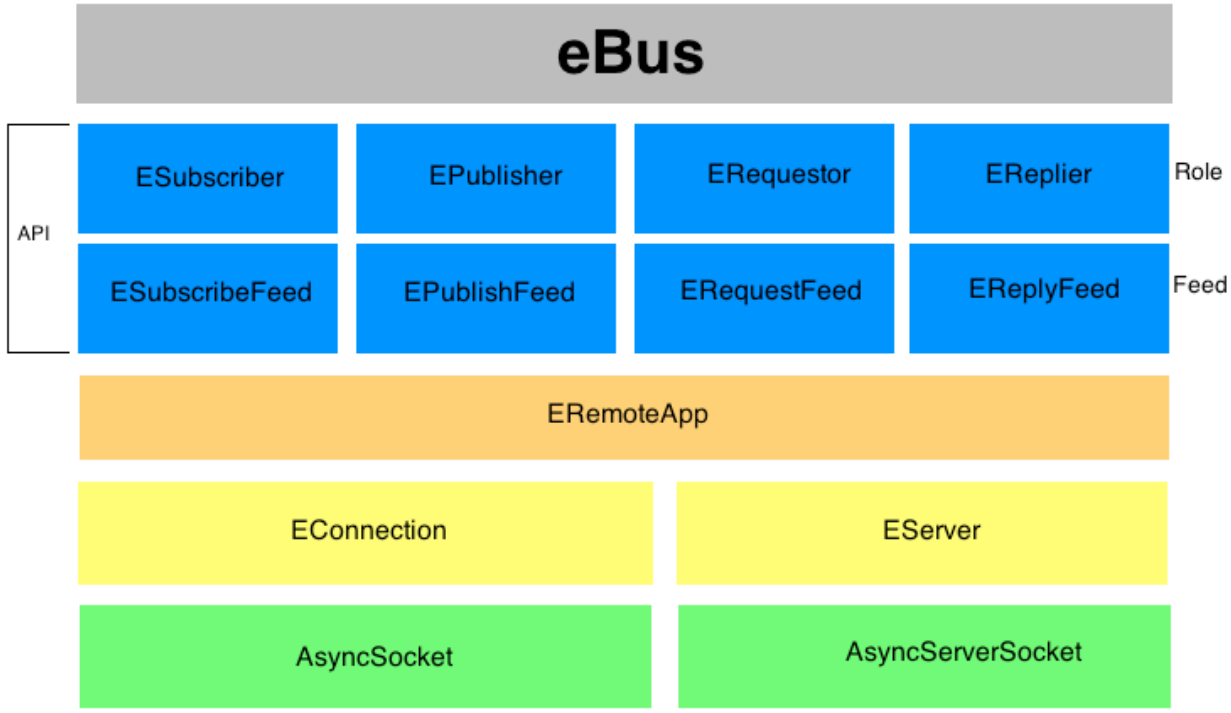
What is eBus?:

- A messaging middleware.
- A message router between objects ...:
  - in the same application,
  - in different applications on the same host, and
  - in different applications on different hosts.
- eBus messages are user-defined Java classes.
- eBus routes messages based on type+topic (type is the message class and topic is the message subject), providing stronger type checking than topic-based routing alone.
- eBus supports publish/subscribe messaging.
- eBus supports request/reply messaging.
- eBus is not a separate message routing process - *it is your process.*

Overview shows how application objects interact with the eBus API to send and receive user-defined messages. An object may implement one or more eBus roles: publisher, subscriber, requestor, and replier.

eBus makes it easy to develop a distributed, message-based application and this manual shows just how simple it is.

Overview



Application classes implement eBus roles and interact with eBus feeds. The publish/subscribe roles are `EPublisher` and `ESubscriber` and their respective feeds are `EPublishFeed` and `ESubscribeFeed`. The request/reply roles are `ERequestor` and `EReplier` using `ERequestFeed` and `EReplyFeed`.

The idea is that a role-playing object interacts with its feed to send and/or receive messages. A publisher only sends messages. A subscriber only receives messages. But a requestor and a replier both send and receive messages.

The next section, "Merrily, We Role Along", shows how to implement eBus roles. Not much to it: you are simply implementing a Java interface. This section also shows how application classes implementing eBus roles interact with eBus feeds.

"Get the Message" section demonstrates how to define a notification, request, and reply message simply be creating a Java class and attaching the necessary annotations.

Section "Connecting Up" describes how eBus applications connect. eBus handles the connection process and message transmission for the application. These connections are invisible to the application code, allowing remote application objects to communicate as if the were in the same JVM.

"Dispatcher" explains how eBus passes messages to application objects while the section "Gentlemen, start your objects" shows how to start up application objects in a thread-safe manner.

Go here to download the latest eBus, if you haven't already.

Go here to see the eBus API documentation.

Merrily, We Role Along.

An application class interacts with eBus by implementing one or more interfaces: `net.sf.eBus.client.EPublisher`, `ESubscriber`, `ERequestor`, and `EReplier`. Each interface works with a specific feed: `net.sf.eBus.client.EPublishFeed`, `ESubscribeFeed`, `ERequestFeed`, and `EReplyFeed`.

The publisher role is demonstrated first.

Please note that an application class may implement more than one eBus interface, even all four. That application class' instance may interact with multiple feeds. A publisher object may publish multiple notification feeds and a subscriber subscribe to multiple feeds. In short, an application class may be very simple - implements one interface and opens one feed, or very complex - implements all four interfaces and opens multiple feeds for each interface. It up to the application developer to decide the complexity level.

Publisher

Step 1: Implementing a publisher

An application can publish messages by first implementing the `net.sf.eBus.client.EPublisher` interface.

Note: Previous code is not shown in subsequent steps. The complete code is show in the final step.

```
import net.sf.eBus.client.EFeedState;
import net.sf.eBus.client.EPublisher;
import net.sf.eBus.client.EPublishFeed;

public class CatalogPublisher
    implements EPublisher
{
    // EPublisher interface has one method.
    @Override public void publishStatus(final EFeedState feedState,
                                        final EPublisherFeed feed) {
         See steps 3 and 5.
    }
}
```

Step 2: Opening and advertising a publisher feed

Applications communicate with eBus using feeds. So the first step is to open a feed, associating it with your `EPublisher` instance:

```
pubFeed = net.sf.eBus.client.EPublishFeed.open(publisher, key, scope)
```

where:

`publisher` is a non-null instance implementing `EPublisher`. Since the publisher object usually opens the publish feed, `this` is passed in as the `publisher` argument.

`key` is a notification message key (meaning that the key's message class is a `net.sf.eBus.messages.ENotificationMessage` subclass).

`scope` is the feed scope.

(Store away the open `EPublishFeed` since this object is used to publish notification messages in step 5 and to retract the advertisement in step 6.)

The second step is to advertise you EPublisher to subscribers:

```
pubFeed.advertise()
```

Note: Do not publish notifications until eBus tells you to start publishing via the `publishStatus` callback (step 3).

Updating the Publisher Feed State

When a publisher calls `EPublishFeed.updateFeedState()` depends on the feed's stability. If the publisher is autonomously generating notification messages, then the feed is stable and the publisher should call `pubFeed.updateFeedState(EFeedState.UP)` immediately after advertising. But if the feed state is dependent on external factors, then the publisher should wait until these factors are determined.

`EPublishFeed.updateFeedState(EFeedState.UP)` *must* be called some time prior to notification publishing. This call may be done any time after advertising and before publishing the first message. If a notification feed is up and the remaining subscriber to that feed unsubscribes, then `publishStatus` is called with `EFeedState.DOWN`. The publisher must then stop posting messages until the feed state comes back up.

It is possible to inform subscribers that the feed is down by unadvertising the feed. When the feed is back up, the feed advertisement is put back in place. The reason for separating `EPublishFeed.advertise()` and `EPublishFeed.updateFeedState(EFeedState)` is due to the cost of removing and restoring an advertisement. It is faster to leave the advertisement in place and simply inform subscribers that the feed is down until further notice.

In this example the publisher autonomously generates notification messages. So it sets its publish feed state to up immediately after advertising.

```
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EFeedState;
import net.sf.eBus.client.EPublisher;
import net.sf.eBus.client.EPublishFeed;
import net.sf.eBus.messages.EMessageKey;

public class CatalogPublisher
    implements EPublisher
{
    public CatalogPublisher(final String subject, final FeedScope scope) {
        mKey = new EMessageKey(CatalogUpdate.class, subject);
        mScope = scope;

        // Set the feed to null to denote that the feed is not yet in place.
        mFeed = null;
    }

    @Override public void startup() {
        try {
            mFeed = EPublishFeed.open(this, mKey, mScope);
            mFeed.advertise();

            // Inform the world that this publisher's feed state is up.
            mFeed.updateFeedState(EFeedState.UP);
        } catch(IllegalArgumentException argex) {
            // Advertisement failed. Place recovery code here.
        }
    }

    @Override public void shutdown() {
         Shown in step 6.
    }

    // Publishes this notification message class/subject key.
    private final EMessageKey mKey;

    // Published messages remain within this scope.
    private final FeedScope mScope;

    // Advertise and publish on this feed.
    private EPublishFeed mFeed;
}
```

Step 3: When to start publishing

eBus tells a publisher when to start publishing its notification messages by calling the `publishStatus` method with the `feedState` parameter set to `EFeedState.UP`. eBus makes this call when there is at least one in-scope subscriber to the notification message key.

```
public class CatalogPublisher
    implements EPublisher
{
    @Override public void publishStatus(final EFeedState feedState,
                                        final EPublishFeed feed) {
        EFeedState publishState;

        // Are we starting a feed?
        if (feedState == EFeedState.UP) {
            // Yes. Start publishing notifications on the feed.
            startPublishing();
        } else {
            // We are stopping the feed.
             More in step 5.
        }

    }

    public void updateProduct(final String productName,
                              final Money price,
                              final int stockQty) {
         Shown in step 4.
    }
}
```

Step 4: Publishing

Publishing a notification message is easy:

1. Instantiate the notification message.
   Note: messages are immutable. Once instantiated the message cannot be modified.

2. Pass the notification message to `EPublishFeed.publish(ENotification message)`.

```
public class CatalogPublisher
    implements EPublisher
{
    public void updateProduct(final String productName,
                              final Money price,
                              final int stockQty) {
        if (mFeed != null && mFeed.isFeedUp() == true) {
            mFeed.publish(
                new CatalogUpdate(productName, price, stockQty));
        }
    }
}
```

Step 5: When to stop publishing

When there are no more subscribers for the published notification message key, eBus call `publishStatus` again with an `EFeedState.DOWN` feed state. At this point, the publisher *must* stop publishing notification messages on that feed. Any further attempts to publish notifications on the feed will result in a thrown `IllegalStateException`.

```
public class CatalogPublisher
    implements EPublisher
{
    @Override public void publishStatus(final EFeedState feedState,
                                        final EPublishFeed feed) {
        // The feed is down until proven otherwise.
        EFeedState publishState;

        // Are we starting a feed? Is the advertisement still in place?
        if (feedState == EFeedState.UP) {
             See step 3.
        } else if (mFeeds.containsKey(key) == true) {
            // This is a request to stop an existing feed.
            stopPublishing();
        }
    }
}
```

Step 6: Unadvertising the publisher

A publisher has three ways to let eBus know that it will no longer be publishing messages on the feed:

1. `EPublishFeed.updateFeedState(EFeedState.DOWN)`: As previously mentioned, this tells eBus that the publisher is temporarily unable to publish notifications on this feed but plans to publish on the feed as soon as the problem is cleared.
2. `EPublishFeed.unadvertise()`: The publisher is retracting the announcement that it can publish notifications on the feed. The publisher is able to put the advertisement back in place on the feed in the future. This scenario could be used where the application is able to enable, disable objects. On enablement, the object advertises the feed. On disablement, the object un-advertises.
3. `EPublishFeed.close()`: The publisher is permanently closing the feed. This also retracts an in place advertisement. Once closed, the feed cannot be used again. If the object intends to use the feed in the future, then a new feed must be opened and that one used.

Note: eBus maintains a weak reference back to application objects using eBus. When eBus detects such an object's finalization, eBus automatically retracts that object's advertisements, subscriptions, and active requests. That said, it is preferable that application objects close their open feeds themselves rather than depending on the eBus automatic retraction.

```
public class CatalogPublisher
    implements EPublisher
{
    // Retract the notification feed.
    @Override public void shutdown() {
        if (mFeed != null) {
            // unadvertise() unnecessary since close() retracts an in-place
            // advertisement.
            mFeed.close();
            mFeed = null;
        }
    }
}
```

Step 7: Complete publisher code

```
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EFeedState;
import net.sf.eBus.client.EPublisher;
import net.sf.eBus.client.EPublishFeed;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ENotificationMessage;

public class CatalogPublisher
    implements EPublisher
{
    public CatalogPublisher(final String subject, final FeedScope scope) {
        mKey = new EMessageKey(CatalogUpdate.class, subject);
        mScope = scope;
        mFeed = null;
    }

    @Override public void startup() {
        try {
            mFeed = EPublishFeed.open(this, mKey, mScope);
            mFeed.advertise();

            // Inform the world that this publisher's feed state is up.
            mFeed.updateFeedState(EFeedState.UP);
        } catch (IllegalArgumentException argex) {
            // Advertisement failed. Place recovery code here.
        }
    }

    @Override public void publishStatus(final EFeedState feedState,
                                        final EPublishFeed feed) {
        EFeedState publishState;

        // Are we starting a feed?
        if (feedState == EFeedState.UP) {
            // Yes. Start publishing notifications on the feed.
            publishState = startPublishing();
        } else {
            // We are stopping the feed.
            stopPublishing();
        }
    }

    public void updateProduct(final String productName,
                              final Money price,
                              final int stockQty) {
        if (mFeed != null && mFeed.isFeedUp() == true) {
            mFeed.publish(
```

```
                new CatalogUpdate(productName, price, stockQty));
        }
    }

    // Retract the notification feed.
    @Override public void shutdown() {
        if (mFeed != null) {
            // unadvertise() unnecessary since close() retracts an in-place
            // advertisement.
            mFeed.close();
            mFeed = null;
        }
    }

    // Starts the notification feed when the feed state is up.
    // Return EFeedState.UP if the notification is successfully started;
    // EFeedState.DOWN if the feed fails to start.
    private EFeedState startPublishing() {
         Application-specific code not shown.
    }

    // Stops the notification feed if up.
    private void stopPublishing() {
         Application-specific code not shown.
    }

    // Publishes this notification message class/subject key.
    private final EMessageKey mKey;

    // Published messages remain within this scope.
    private final FeedScope mScope;

    // Advertise and publish on this feed.
    private EPublishFeed mFeed;
}
```

Subscriber

Step 1: Implementing a subscriber

Every application class receiving notification messages must implement `net.sf.eBus.client.ESubscriber` interface.

Go here to learn how eBus calls back to clients.

Note: previous code is not show in subsequent steps. Go here to see the complete code.

```
import net.sf.eBus.client.EFeedState;
import net.sf.eBus.client.ESubscribeFeed;
import net.sf.eBus.client.ESubscriber;
import net.sf.eBus.messages.ENotificationMessage;

public class CatalogSubscriber
    implements ESubscriber
{
    // ESubscriber interface has two methods.
    @Override public void feedStatus(final EFeedState feedState,
                                     final ESubscribeFeed feed) {
        See step 3.
    }

    @Override public void notify(final ENotificationMessage msg,
                                 final ESubscribeFeed feed) {
        See step 4.
    }
}
```

Step 2: Subscribing to a notification message and subject

Subscribing tells eBus which messages and subjects a subscriber wants to receive. This is done in two steps.

The first step is opening the subscription feed:

```
subFeed = net.sf.eBus.client.ESubscribeFeed.open(subscriber, key, scope, condition);
```

where:

`subscriber` is a non-null instance implementing `ESubscriber`. Since the subscriber object usually opens the feed itself, `this` is passed in as the `subscriber` argument.

`key` is the message key containing the subscribed notification message class and message subject.

`scope` is the feed scope.

`condition` is the optional condition used to restrict delivered notification messages to those which satisfy the condition. This argument may be `null`.

The second step puts the subscription in place:

```
subFeed.subscribe();
```

```
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EFeedState;
import net.sf.eBus.client.ESubscribeFeed;
import net.sf.eBus.client.ESubscriber;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ENotificationMessage;

public class CatalogSubscriber
    implements ESubscriber
{
    public CatalogSubscriber(final String subject, final FeedScope scope) {
        mKey = new EMessageKey(CatalogUpdate.class, subject);
        mScope = scope;
        mFeed = null;
    }

    @Override public void startup() {
        try {
            // This subscription has no associated ECondition; passing in null.
            mFeed = ESubscribeFeed.open(this, mKey, mScope, null);
            mFeed.subscribe();
        } catch(IllegalArgumentException argex) {
            // Feed open failed. Place recovery code here.
        }
    }

    @Override public void feedStatus(final EFeedState feedState,
                                     final ESubscribeFeed feed) {
         See step 3.
    }

    @Override public void notify(final ENotificationMessage msg,
                                 final ESubscribeFeed feed) {
         See step 4.
    }

    @Override public void shutdown() {
         See step 5.
    }

    // Subscribe to this notification message class/subject key and feed scope.
    private final EMessageKey mKey;
    private final FeedScope mScope;

    // Store the feed here so it can be used to unsubscribe.
    private ESubscribeFeed mFeed;
}
```

Step 3: Handling a publisher feed status

eBus informs a subscriber whether there are any publishers actively publishing messages to the subscribed notification message and subject. The feed state is set to `EFeedState.UP` if there is at least one publisher in scope able to publish the message. At this point you may expect notification messages to be delivered to your notify method (although there is no guarantee that the publisher has anything to send).

If there no publishers able to provide the requested notification message key, eBus will call the feed status method with a `EFeedState.DOWN` feed state. At this point you will not receive any calls to your notify method for the specified feed.[1]

[1] If a subscriber is subscribed to multiple feeds, the fact that one of the feeds is down does not preclude the subscriber from receiving notifications for the other feeds.

```
public class CatalogSubscriber
    implements ESubscriber
{
    @Override public void feedStatus(final EFeedState feedState,
                                     final ESubscribeFeed feed) {
        // What is the publisher feed state?
        if (feedState == EFeedState.DOWN) {
            // Down. There are no publishers. Expect no notifications until a
            // publisher is found. Put error recovery code here.
        } else {
            // Up. There is at least one publisher. Expect to receive notifications.
        }
    }
}
```

Step 4: Handling notifications

When there is at least one publisher with a `EFeedState.UP` publish feed status, you may expect notification messages to be delivered to your notify method (although there is no guarantee that the publisher has anything to send).

```
public class CatalogSubscriber
    implements ESubscriber
{
    @Override public void notify(final ENotificationMessage msg,
                                 final ESubscribeFeed feed) {
        Notification handling code here.
    }
}
```

Step 5: Unsubscribing

A subscriber has two ways to retract a subscription:

1. `ESubscribeFeed.unsubscribe()`: This call retracts the subscription but leaves the feed open. This may be desirable if application objects can be enabled and disabled. On enablement the subscription is put in place and retracted on disablement.

2. `ESubscribeFeed.close`: This permanently closes the subscription feed. Once closed the feed can no longer be used. This call retracts an in place subscription. This may be desirable when shutting down an object.

Note: you may still receive a notify callback after unsubscribing because the message was about to be delivered when unsubscribing. So it may be necessary to check if the subscription is in place before processing the delivered notification message.

Note: eBus maintains a weak reference back to application objects using eBus. When eBus detects such an object’s finalization, eBus automatically retracts that object’s advertisements, subscriptions, and active requests. That said, it is preferable that application objects close their open feeds themselves rather than depending on the eBus automatic retraction.

```
public class CatalogSubscriber
    implements ESubscriber
{
    @Override public void shutdown() {
        if (mFeed != null) {
            // mFeed.unsubscribe() is not necessary since close() will unsubscribe.
            mFeed.close();
            mFeed = null;
        }
    }
}
```

Step 6: Complete subscriber code

```
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EFeedState;
import net.sf.eBus.client.ESubscribeFeed;
import net.sf.eBus.client.ESubscriber;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ENotificationMessage;

public class CatalogSubscriber
    implements ESubscriber
{
    public CatalogSubscriber(final String subject, final FeedScope scope) {
        mKey = new EMessageKey(CatalogUpdate.class, subject);
        mScope = scope;
        mFeed = null;
    }

    @Override public void startup() {
        try {
            // This subscription has no associated ECondition; passing in null.
            mFeed = ESubscribeFeed.open(this, mKey, mScope, null);
            mFeed.subscribe();
        } catch(IllegalArgumentException argex) {
            // Feed open failed. Place recovery code here.
        }
    }

    @Override public void feedStatus(final EFeedState feedState,
                                     final ESubscribeFeed feed) {
        // What is the feed state?
        if (feedState == EFeedState.DOWN) {
            // Down. There are no publishers. Expect no notifications until a
            // publisher is found. Put error recovery code here.
        } else {
            // Up. There is at least one publisher. Expect to receive notifications.
        }
    }

    @Override public void notify(final ENotificationMessage msg,
                                 final ESubscribeFeed feed) {
        Notification handling code here.
    }

    @Override public void shutdown() {
        // mFeed.unsubscribe() is not necessary since close() will unsubscribe.
        mFeed.close();
        mFeed = null;
```

```
    }

    // Subscribe to this notification message class/subject key and feed scope.
    private final EMessageKey mKey;
    private final FeedScope mScope;

    // Store the feed here so it can be used to unsubscribe.
    private ESubscribeFeed mFeed;
}
```

Replier

Step 1: Implementing a replier

Every application class that wants to reply to request messages must implement the `net.sf.eBus.client.EReplier` interface. Repliers both receive request messages and send reply messages.

Go here to learn how eBus calls back to clients.

Note: previous code is not show in subsequent steps. Go here to see the complete code.

```
import net.sf.eBus.client.EReplier;
import net.sf.eBus.client.EReplyFeed;
import net.sf.eBus.messages.ERequestMessage;

public class CatalogReplier
    implements EReplier
{
    // EReplier interface has two methods.
    @Override public void request(final EReplyFeed.ERequest request,
                                  final EReplyFeed feed) {
        See step 3.
    }

    @Override public void cancelRequest(final EReplyFeed. ERequest request,
                                        final EReplyFeed feed) {
        See step 4.
    }
}
```

Step 2: Advertising a replier

Advertising tells eBus about what request messages a replier can handle. A replier will not receive requests until it first advertises the request message key by opening the reply feed and then advertising the replier:

```
replyFeed = net.sf.eBus.client.EReplyFeed.open(replier,key,scope,condition);
```

where:

`replier` is the required `EReplier` instance. Since replier objects usually open their own reply feeds, `this` is passed in as the `replier` argument.

`key` is a request message key containing the key's `ERequestMessage` subject and message subject.

`scope` is the feed scope.

`condition` is an optional condition used to restrict delivered request messages to those which satisfy the condition. May be `null`.

Once opened, advertise the replier to requestors:

```
replyFeed.advertise();
replyFeed.updateFeedState(EFeedState.UP);
```

The `EReplyFeed.updateFeedState(EFeedState.UP)` informs eBus that this replier can respond to requests. Just like `EPublishFeed`, there is a separation between advertising a feed and providing a feed. The idea here is that if a replier is dependent on a resource need to generate responses and that resource is unavailable, then the replier calls `EReplyFeed.updateFeedState(EFeedState.DOWN)`. This informs requestors that the replier is unavailable to handle requests until further notice. When the resource is again available, then the replier calls `EReplyFeed.updateFeedState(EFeedState.UP)` and requests will again be sent to the replier.

```
import java.util.ArrayList;
import java.util.List;
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EReplier;
import net.sf.eBus.client.EReplyFeed;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ERequestMessage;

public class CatalogReplier
    implements EReplier
{
    public CatalogReplier(final String subject, final FeedScope scope) {
        mKey = new EMessageKey(com.acme.CatalogOrder.class, subject);
        mScope = scope;
        mFeed = null;
        mRequests = new ArrayList<>();
    }

    @Override public void startup() {
        try {
            // This advertisement has no associated ECondition; passing in null.
            mFeed = EReplyFeed.open(this, mKey, mScope, null);
            mFeed.advertise();

            // Let requestors know that this replier responds to requests.
            mFeed.updateFeedState(EFeedState.UP);
        } catch (IllegalArgumentException argex) {
            // Advertisement failed. Place recovery code here.
        }
    }

    @Override public void shutdown() {
         See step 6.
    }

    // Replies to this request message class/subject key.
    private final EMessageKey mKey;

    // Replier handles requests posted within this scope.
    private final FeedScope mScope;

    // Store the replier feed here so it can be used to unadvertise.
    private EReplyFeed mFeed;

    // Stores the active requests. See steps 3 and 4.
    private final List<EReplyFeed.ERequest> mRequests;
}
```

Step 3: Handling a request

When eBus forwards a request message to a matching replier, the replier can respond either synchronously or asynchronously. A synchronous response means replying within the request callback method. An asynchronous response means calling `ERequest` after returning from the request callback.

The request message is stored in the `ERequest` instance and can be retrieved by calling `ERequest.request()`.

```
import net.sf.eBus.messages.EReplyMessage;
import net.sf.eBus.messages.EReplyMessage.ReplyStatus;

public class CatalogReplier
    implements EReplier
{
    @Override public void request(final EReplyFeed.ERequest request,
                                  final EReplyFeed feed) {
        // The request message is stored inside the request.
        final ERequestMessage msg = request.request();

        try {
            // Start processing the request now and reply later.
            // When sending multiple replies, set the reply status to
            // ReplyStatus.OK_CONTINUING.
            // For the final reply, set the reply status to ReplyStatus.OK_FINAL.
            startOrderProcessing(msg, request);
            mRequests.add(request);
        } catch (Exception jex) {
            // Exception thrown by startOrderProcessing().
            final EReplyMessage reply =
                new CatalogOrderReply(
                    ReplyStatus.ERROR, // reply status.
                    jex.getMessage()); // reply reason.

            request.reply(reply);
        }
    }

    @Override public void cancelRequest(final EReplyFeed.ERequest request,
                                        final EReplyFeed feed) {
         See step 4.
    }

    public void orderReply(final ERequest request,
                           final boolean status,
                           final String reason) {
         See step 5.
    }

    @Override public void shutdown() {
         See step 6.
    }
}
```

Step 4: Canceling a request

When `cancelRequest` is called, the requestor is terminating the request. The replier is informed of the cancellation via the callback:

```
ERequestor.cancelRequest(ERequestFeed.ERequest, EReplyFeed);
```

While the replier may not be able to stop the in-progress request processing, no more replies to request will be accepted.

```
import net.sf.eBus.client.ERequest.CancelStatus;

public class CatalogReplier
    implements EReplier
{
    @Override public void cancelRequest(final EReplyFeed.ERequest request,
                                        final EReplyFeed feed) {
        // Is this request still active? It is if the request is listed.
        if (mRequests.remove(request) == true) {
            // Yes, try to stop the request processing.
            try {
                // Throws an exception if the request cannot be stopped.
                stopOrderProcessing(request)
            } catch (Exception jex) {
                // Ignore since nothing else can be done.
            }
        }
    }

    public void orderReply(final EReplyFeed.ERequest request,
                           final boolean status,
                           final String reason) {
         See step 5.
    }

    @Override public void shutdown() {
         See step 6.
    }
}
```

Step 5: Replying to a request

eBus allows a replier to send multiple reply messages for a given request. If you want to send an intermediate reply, set the status to `EReplyMessage.ReplyStatus.OK_CONTINUING`. For the final reply, set the status to `EReplyMessage.ReplyStatus.OK_FINAL`. If an error occurs which precludes successfully completing the request, set the status to `EReplyMessage.ReplyStatus.ERROR`. This reply status may be sent even after intermediate replies were previously sent.

An `ERROR` or `OK_FINAL` reply is a final reply. No more replies may be sent after sending either status.

```
public class CatalogReplier
    implements EReplier
{
    // Send an asynchronous reply to the request.
    public void orderReply(final EReplyFeed.ERequest request,
                           final ReplyStatus status,
                           final String reason) {
        final ERequestAd ad = mRequests.get(request);

        if (mRequests.contains(request) == true && request.isActive() == true) {
            request.reply(new OrderReply(status, reason), ad);

            // If the request processing is complete, remove the request.
            if (status.isFinal() == true) {
                mRequests.remove(request);
            }
        }
    }

    @Override public void shutdown() {
         See step 6.
    }
}
```

Step 6: Unadvertising a replier

A relier has three ways to let eBus know that it will no longer accept requests on the feed:

1. `EReplyFeed.updateFeedState(EFeedState.DOWN)`: As previously mentioned, this tells eBus that the replier is temporarily unable to handle requests on this feed but plans to do so again on the feed as soon as the problem is cleared.
2. `EReplyFeed.unadvertise()`: The replier is retracting the announcement that it can respond to requests on the feed. Un-advertising automatically sends an `EReplyMessage` with an error reply status to all active request on the feed. The replier is able to put the advertisement back in place on the feed in the future. This scenario could be used where the application is able to enable, disable objects. On enablement, the object advertises the feed. On disablement, the object un-advertises.
3. `EReplyFeed.close()`: The replier is permanently closing the feed. This also retracts an in place advertisement. Once closed, the feed cannot be used again. Like `unadverttise`, the feed's active requests receive an error reply.

Note: eBus maintains a weak reference back to application objects using eBus. When eBus detects such an object's finalization, eBus automatically retracts that object's advertisements, subscriptions, and active requests. That said, it is preferable that application objects close their open feeds themselves rather than depending on the eBus automatic retraction.

```
public class CatalogReplier
    implements EReplier
{
    @Override public void shutdown() {
        final String subject = mKey.subject();
        EReplyMessage reply;

        // While eBus will does this for us, it is better to do it ourselves.
        for (EReplyFeed.ERequest request : mRequests) {
            reply = new EReplyMessage(subject, ReplyStatus.ERROR, "shutting down");
            request.reply(reply);
        }

        mRequests.clear();

        if (mFeed != null) {
            // close() implicitly unadvertises the feed.
            mFeed.close();
            mFeed = null;
        }
    }
}
```

Step 7: Complete replier code

```
import java.util.ArrayList;
import java.util.List;
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EReplier;
import net.sf.eBus.client.EReplyFeed;
import net.sf.eBus.messages.EReplyMessage
import net.sf.eBus.messages.EReplyMessage.ReplyStatus;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ERequestMessage;

public class CatalogReplier
    implements EReplier
{
    public CatalogReplier(final String subject, final FeedScope scope) {
        mKey = new EMessageKey(com.acme.CatalogOrder.class, subject);
        mScope = scope;
        mFeed = null;
        mRequests = new ArrayList<>();
    }

    @Override public void startup() {
        try {
            // This advertisement has no associated ECondition; passing in null.
            mFeed = EReplyFeed.open(this, mKey, mScope, null);
            mFeed.advertise();
            mFeed.updateFeedState(EFeedState.UP);
        } catch (IllegalArgumentException argex) {
            // Advertisement failed. Place recovery code here.
        }
    }

    @Override public void request(final EReplyFeed. ERequest request,
                                  final EReplyFeed feed) {
        final ERequestMessage msg = request.request();

        try {
            mRequests.add(request);
            startOrderProcessing(msg, request);
        } catch (Exception jex) {
            request.reply(new CatalogOrderReply(ReplyStatus.ERROR,  // reply status.
                                                jex.getMessage())); // reply reason.
        }
    }

    @Override public void cancelRequest(final EReplyFeed. ERequest request,
                                        final EReplyFeed feed) {
```

```
        // Is this request still active? It is if the request is listed.
        if (mRequests.remove(request) == true) {
            // Yes, try to stop the request processing.
            try {
                // Throws an exception if the request cannot be stopped.
                stopOrderProcessing(request)
            } catch (Exception jex) {
                // Ignore since nothing else can be done.
            }
        }
    }

    public void orderReply(final EReplyFeed.ERequest request,
                           final boolean status,
                           final String reason) {
        final ERequestAd ad = mRequests.get(request);

        if (mRequests.contains(request) == true && request.isActive() == true) {
            request.reply(new OrderReply(status, reason), ad);

            // If the request processing is complete, remove the request.
            if (status.isFinal() == true) {
                mRequests.remove(request);
            }
        }
    }

    @Override public void shutdown() {
        final String subject = mKey.subject();
        EReplyMessage reply;

        // While eBus will does this for us, it is better to do it ourselves.
        for (EReplyFeed.ERequest request : mRequests) {
            reply = new EReplyMessage(subject, ReplyStatus.ERROR, "shutting down");
            request.reply(reply);
        }

        mRequests.clear();

        if (mFeed != null) {
            mFeed.close();
            mFeed = null;
        }
    }

    private final EMessageKey mKey;
    private final FeedScope mScope;
    private EReplyFeed mFeed;
    private final List< EReplyFeed. ERequest> mRequests;
}
```

Requestor

Step 1: Implementing a requestor

Every application class that sends request messages must implement the `net.sf.eBus.client.ERequestor` interface.

Go here to learn how eBus calls back to clients.

Note: previous code is not show in subsequent steps. Go here to see the complete code.

```
import net.sf.eBus.client.ERequestFeed;
import net.sf.eBus.client.ERequestor;
import net.sf.eBus.messages.EReplyMessage;

public class CatalogRequestor
    implements ERequestor
{
    // ERequestor interface has two methods.
    @Override public void feedStatus(final EFeedState feedState,
                                     final ERequestFeed feed) {
        See step 3.
    }

    @Override public void reply(final int remaining,
                                final EReplyMessage reply,
                                final ERequestFeed.ERequest request) {
        See step 5.
    }
}
```

Step 2: Opening a request

A requestor makes requests by opening the request feed and subscribing.

```
requestFeed = net.sf.eBus.client.ERequestFeed.open(requestor, key, scope);
requestFeed.subscribe();
```

where:

`requestor` is the required `ERequestor` instance.

`key` is a request message key containing the key's `ERequestMessage` subject and message subject.

`scope` is the feed scope.

```
import java.util.ArrayList;
import java.util.List;
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.ERequestFeed;
import net.sf.eBus.client.ERequestor;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.EReplyMessage;

public class CatalogRequestor
    implements ERequestor
{
    public CatalogRequestor(final String subject, final FeedScope scope) {
        mKey = new EMessageKey(com.acme.CatalogOrder.class, subject);
        mScope = scope;
        mFeed = null;
        mRequests = new ArrayList<>();
    }

    @Override public void startup() {
        try {
            // Put the request advertisement in place.
            mFeed = ERequestFeed.open(this, mKey, mScope);
            mFeed.subscribe();
        } catch (IllegalArgumentException argex) {
            // Open failed. Place recovery code here.
        }
    }

    @Override public void shutdown() {
         See step 6.
    }

    @Override public void reply(final int remaining,
                                final EReplyMessage reply,
                                final ERequestFeed.ERequest request) {
         See step 4.
    }

    private final EMessageKey mKey;
    private final FeedScope mScope;
    private final List<ERequestFeed.ERequest> mRequests;
    private ERequestFeed mFeed;
}
```

Step 3: Handling a replier's feed status

eBus informs a requestor whether there are any repliers for the request message and subject. The feed state is set to `EFeedState.UP` if there is at least one replier. At this point you may expect to successfully place a request. There is alway a possibility that the request will fail due to:

1. all in-scope repliers having an advertisement condition and
2. the request fails each of those conditions.

If there no repliers for a request message key, eBus will call the feed status method with a `EFeedState.DOWN` feed state. At this point you can expect requests to fail.

```
public class CatalogRequestor
    implements ERequestor
{
    @Override public void feedStatus(final EFeedState feedState,
                                     final ERequestFeed feed) {
        // What is the replier feed state?
        if (feedState == EFeedState.DOWN) {
            // Down. There are no repliers. Requests should fail until an UP feed
            // state is announced.
        } else {
            // Up. There is at least one replier. Requests may now be placed.
        }
    }
}
```

Step 4: Making a request

Requests are placed using the open and subscribed `ERequestFeed`. Create an `ERequestMessage`-subclassed message and have the request feed send it:

```
final ERequestMessage msg = new AppRequestMessage();
final ERequestFeed.ERequest request = mFeed.request(msg);
```

`ERequestFeed.request(ERequestMessage)` returns the feed instance encapsulating the request. The requestor may now expect to receive replies to this request.

If there are no repliers for the request and feed scope, then an `IllegalStateException` is thrown. No replies will be sent to the requestor for this request message.

Note: once the `ERequestFeed.ERequest` instance is returned, the *application* is responsible for tracking all active requests. The `ERequestFeed` does *not* store or track requests on behalf of the application. `ERequestFeed.close()` does *not* automatically close active requests opened by the `ERequestFeed` instance. The application is responsible for closing active requests.

Note: if a request is made on a non-eBus Dispatcher thread, then it is important to synchronize placing requests with receiving replies. There is a very real chance that a reply will be received *before* `ERequestFeed.request()` returns. It is recommended that the application store away all information needed to handle a reply before making the request. Any bookkeeping that cannot be done prior to making the request means that the request and reply handling must be made inside some sort of synchronization.

If a request is made on an eBus Dispatcher thread, then synchronization is not necessary.

```
public class CatalogRequestor
    implements ERequestor
{
    public void placeOrder(final String product,
                           final int quantity,
                           final Price price,
                           final ShippingEnum shipping,
                           final ShippingAddress address)
    {
        final CatalogOrder msg =
            new CatalogOrder(product, quantity, price, shipping, address);

        try {
            // Do any application-specified bookkeeping here before placing the
            // request. This way the information needed to handle the reply is
            // in place.
            // Order placed via a non-eBus thread, so synchronization is needed.
            // You need to synchronize this requestor because there is a real chance
            // that eBus will call reply() before returning from request() and this
            // call is made from a non-eBus thread (go here to learn more).
            synchronized (mRequests) {
                mRequests.add(mFeed.request(msg));
            }
        } catch (Exception jex) {
            // Request failed. Put recovery code here.
        }
    }

    @Override public void shutdown() {
         See step 5.
    }

    @Override public void reply(final int remaining,
                                final EReplyMessage reply,
                                final ERequestFeed.ERequest request) {
         See step 4.
    }
}
```

Step 5: Receiving replies

eBus continues to send replies to the requestor as long as repliers are sending replies. The `ERequest.reply(int remaining, EReplyMessage reply, ERequestFeed.ERequest request)` callback parameters are:

- `remaining`: the number of repliers still sending replies. When this value is zero, then this is the final reply and no more replies will be forthcoming. The request is completed.
- `reply`: the reply message itself. Check `reply.replyStatus` to determine if 1) the request is accepted or rejected and 2) if this is the final reply *from this replier* or if more replies are forthcoming. If `replyStatus` is `ReplyStatus.ERROR`, then the replier is rejecting the request and this is the replier's final reply. `reply.replyReason` should contain text explaining why the request was rejected. If `replyStatus` is `ReplyStatus.OK_CONTINUING`, then the replier accepted the request and sent this reply with more to follow. `ReplyStatus.OK_FINAL` means that this is the replier's final reply to the request.

  Note: it is possible for a replier to send an initial `ReplyStatus.OK_CONTINUING` reply followed by a `ReplyStatus.ERROR` reply due to the replier being unable to complete the request.
- `request`: this reply applies to this request.

Again, synchronization may be needed to coordinate the request with the reply.

```
public class CatalogRequestor
    implements ERequestor
{
    @Override public void shutdown() {
         See step 6.
    }

    @Override public void reply(final int remaining,
                                final EReplyMessage reply,
                                final ERequestFeed.ERequest request) {
        final String reason = msg.replyReason();

        if (msg.replyStatus == EReplyMessage.ReplyStatus.ERROR) {
            // The replier rejected the request. Report the reason
        }
        // The replier accepted the request. Is this the last reply?
        else if (msg.replyStatus == EReplyMessage.ReplyStatus.OK_CONTINUING) {
            // The replier will be sending more replies.
        } else {
            // This is the replier's last reply.
        }

        // Have all replies been received from all repliers?
        if (remaining == 0) {
            // Yes. Remove the request from the active list.
            synchronized (mRequests) {
                mRequests.remove(request);
            }
        }
    }
}
```

Step 6: Canceling a request

An application is free to cancel an active request at any time. This is done by closing the returned ERequestFeed.ERequest instance:

```
ERequestFeed.ERequest.close();
```

It is still possible for the requestor to receive replies for this request after calling `close()` because the replies were posted to the requestor prior to the close but not yet delivered.

Closing an inactive or completed request is harmless.

```
public class CatalogRequestor
    implements ERequestor
{
    // Cancel all outstanding active requests and close the request feed.
    @Override public void shutdown() {
        synchronized (mRequests) {
            for (ERequestFeed.ERequest request : mRequests) {
                request.close();
            }

            mRequests.clear();
        }

        if (mFeed != null) {
            mFeed.close();
            mFeed = null;
        }
    }
}
```

Step 7: Complete requestor code

```
import java.util.ArrayList;
import java.util.List;
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.ERequestFeed;
import net.sf.eBus.client.ERequestor;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.EReplyMessage;

public class CatalogRequestor
    implements ERequestor
{
    public CatalogRequestor(final String subject, final FeedScope scope) {
        mKey = new EMessageKey(com.acme.CatalogOrder.class, subject);
        mScope = scope;
        mFeed = null;
        mRequests = new ArrayList<>();
    }

    @Override public void startup() {
        try {
            mFeed = ERequestFeed.open(this, mKey, mScope);
            mFeed.subscribe();
        } catch (IllegalArgumentException argex) {
            // Open failed. Place recovery code here.
        }
    }

    @Override public void shutdown() {
         synchronized (mRequests) {
            for (ERequestFeed.ERequest request : mRequests) {
                request.close();
            }

            mRequests.clear();
        }

        if (mFeed != null) {
            mFeed.close();
            mFeed = null;
        }
    }

    @Override public void feedStatus(final EFeedState feedState,
                                     final ERequestFeed feed) {
        if (feedState == EFeedState.DOWN) { // Down. There are no repliers.
        } else { /* Up. There is at least one replier. */ }
    }
```

```
    @Override public void reply(final int remaining,
                                final EReplyMessage reply,
                                final ERequestFeed.ERequest request) {
        final String reason = msg.replyReason();

        if (msg.replyStatus == EReplyMessage.ReplyStatus.ERROR) {
            // The replier rejected the request. Report the reason
        } else if (msg.replyStatus == EReplyMessage.ReplyStatus.OK_CONTINUING) {
            // The replier will be sending more replies.
        } else {
            // This is the replier's last reply.
        }

        if (remaining == 0) {
            synchronized (mRequests) {
                mRequests.remove(request);
            }
        }
    }

    public void placeOrder(final String product,
                           final int quantity,
                           final Price price,
                           final ShippingEnum shipping,
                           final ShippingAddress address) {
        final CatalogOrder msg =
            new CatalogOrder(product, quantity, price, shipping, address);

        try {
            synchronized (mRequests) {
                mRequests.add(mFeed.request(msg));
            }
        } catch (Exception jex) {
            // Request failed. Put recovery code here.
        }
    }

    private final EMessageKey mKey;
    private final FeedScope mScope;
    private final List<ERequestFeed.ERequest> mRequests;
    private ERequestFeed mFeed;
}
```

Using Lambda Expression Callbacks

eBus v. 4.2.0 introduced using lambda expression-based callbacks rather than by overriding the role interface methods. This was accomplished by giving the role interface methods a default implementation which throws an `UnsupportedOperationException`. An application class is still required to implement the role interface associated with a feed but not required to override the interface methods.

Instead, the class calls the feed's callback method, passing in a lambda expression which defines the callback target. eBus calls back to this code rather than the role interface method. Note that this callback must be put in place after the feed is opened but before it is advertised/subscribed. Setting a callback when the feed is closed or advertised/subscribed results in an `IllegalStateException`. Further, if the application neither overrides the role interface method nor puts the matching callback in place, then `advertise()` and `subscribe()` throw an `IllegalStateException` which explains that eBus has no way to call back to the application.

It is possible to mix-and-match interface method override with a callback. The next page show how a class implementing `ESubscriber` receives feed status callbacks using the interface method and notification messages using a lambda expression callback. This example also shows the class opening two different feeds, which is the reason for creating lambda expression callbacks.

The subscriber needs to handle multiple notification feeds but each in a unique way. Using a role interface means that notifications from those different feeds all arrive at the same place:

```
notify(ENotificationMessage, ESubscribeFeed).
```

This method becomes a message router, untangling the different notification messages and forwarding each message to its ultimate destination. This method is pure overhead.

This overhead is removed by using `ESubscribeFeed.notifyCallback(NotifyCallback)` to directly link eBus with that ultimate destination. The following example code is taken from the previous `ESubscriber` code. So much of the extraneous code is elided to focus attention on the lambda callbacks.

Note: lambda expression callbacks are backward compatible with previous eBus code. Applications will see no performance degradation using eBus 4.2.0 or later.

```
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EFeedState;
import net.sf.eBus.client.ESubscribeFeed;
import net.sf.eBus.client.ESubscriber;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ENotificationMessage;

public class CatalogSubscriber
    implements ESubscriber
{
    public CatalogSubscriber(final String subject, final FeedScope scope) { ... }

    @Override public void startup() {
        try {
            // ECondition may now be defined using a lambda expression.
            mFeed1 = ESubscribeFeed.open(this,
                                         mKey1,
                                         mScope1,
                                         (m -> ((AppMessage) m).value >= 100));
            mFeed1.notifyCallback((msg, feed) ->
            { /* Put mFeed1 notification handling code here */ });
            mFeed1.subscribe();

            mFeed2 = ESubscribeFeed.open(this, mKey2, mScope2, null);
            mFeed2.notifyCallback(this::latestUpdate);
            mFeed2.subscribe()
        } catch (IllegalArgumentException argex) {
            // Feed open failed. Place recovery code here.
            // Note: mFeed is still null.
        }
    }

    @Override public void feedStatus(final EFeedState feedState,
                                     final ESubscribeFeed feed) {
        // Status updates for feeds handled in the same way.
    }

    private void latestUpdate(final ENotificationMessage msg,
                              final ESubscribeFeed feed) {
        // mFeed2 notification handling code here.
        // Note: method has same signature as ESubscribe.notify method.
    }
}
```

ERequestor Callbacks

eBus request message classes specify which reply messages may be sent in response to the request. Given that, it would be nice to associate a different callback for each reply message class. With that in mind, eBus 4.3.2 introduced a new request callback method:

```
public void requestCallback(EMessageKey, ReplyCallback)
```

The idea is that the callback is associated with a particular reply message key. A reply is forwarded to the callback associated with the reply's message key.

Note: care must be taken when using both `requestCallback(ReplyCallback)` and `requestCallback(EMessageKey, ReplyCallback)`. The first method sets the callback for all reply message keys *and will overwrite any previously specified reply message key callbacks*. Therefore, you should use the first method to set a generic callback and then the second method for reply-specific callbacks.

For example, take the following request message class:

```
@EFieldInfo (fields ={...})
@EReplyInfo(replyMessageClasses={OptionOrderState.class,OptionOrderFill.class})
public final class OptionOrderRequest
    extends ERequestMessage
```

`OptionOrderRequest` has three possible replies: `EReplyMessage`, `OptionOrderState`, and `OptionOrderFill`. When an option order is placed, `EReplyMessage` is sent in response to specify whether it is accepted (`EReplyMessage.ReplyStatus.OK_CONTINUING`) or rejected (`EReplyMessage.ReplyStatus.ERROR`). If accepted, then one or more `OptionOrderState` messages is sent until either the order is completely filled or canceled (where `isFinal()` returns `true`). The `OptionOrderFill` specifies that either a partial or complete fill was made against the order. This message is never a final reply (i.e., reply state is always `OK_CONTINUING`).

`EReplyMessage` is handled by a generic method `orderReply` and the last two by separate methods `orderState` and `orderFill`. This can be done as follows:

```
final String option = "...";
final EMessageKey requestKey = new EMessageKey(OptionOrderRequest.class, option);
final EMessageKey stateKey = new EMessageKey(OptionOrderState.class, option);
final EMessageKey fillKey = new EMessageKey(OptionOrderFill.class, option);

orderFeed = ERequestFeed.open(this, requestKey, EFeed.FeedScope.REMOTE);

// The *request* status is handled separately from *order* status.
orderFeed.statusCallback(this::requestStatus);

// Put the generic order reply handler in place first.
orderFeed.replyCallback(this::orderReply);

// Secondly, put the specific reply message callbacks in place using the eBus 4.3.2
// replyCallback method.
orderFeed.replyCallback(stateKey, this::orderState);
orderFeed.replyCallback(fillKey, this::orderFill);
```

Get the Message

Messages are what eBus is all about. Messages are easy to define because they are the simplest of POJOs[2]. Beyond constructors, an eBus message class does not require any methods or fields (although a field-less message is not very useful). But, like any Java class definition, a message may be as complex as you wish to make it.

This section shows how to define eBus message classes.

[2] POJO: Plain, Old Java Object.

Step 0: eBus supported message field types

eBus message fields must either an eBus-defined type or a user-defined type (shown in step 5). The eBus-defined types are:

1. a Java primitive (`boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`),
2. the Java class equivalent to the above primitives,
3. an `enum` type,
4. `java.math.BigDecimal` and `BigInteger`,
5. `java.lang.Class`,
6. `java.util.Date`,
7. `java.time` classes `Duration`, `Instant`, `LocalDate`, `LocalTime`, `LocalDateTime`, `MonthDay`, `OffsetTime`, `OffsetDateTime`, `Period`, `YearMonth`, `ZoneOffset`, `ZoneId`, and `ZonedDateTime`,
8. `java.io.File`,
9. `java.net.InetAddress`,
10. `java.net.InetSocketAddress`,
11. `net.sf.eBus.messages.EMessageKey`,
12. `java.lang.String`,
13. `java.net.URI`,
14. `net.sf.eBus.messages.EFieldList`[3], and
15. `net.sf.eBus.messages.EMessageList`.

A homogenous array of the above types by appending `[]` to the end of the message type.

[3] See section "Arrays and List Fields" for more information about field lists, message lists, and arrays.

Step 1: Message Class

eBus messages are Java classes extending either `net.sf.eBus.messages.ENotification`, `ERequestMessage`, or `EReplyMessage`, and having zero to 31[4] `public final` data members:

```
import java.io.Serializable;
import net.sf.eBus.messages.ENotificationMessage;

 See step 2.
public final class CatalogOrder
    extends ERequestMessage
    implements Serializable
{
     See step 3.

    // All transported data members must be public, final, and eBus-supported types.
    // Note: Money is a net.sf.eBus.message.EField subclass which stores the price
    // and currency.
    public final String orderId;
    public final Money price;  See step 5.
    public final int quantity;
    private static final long serialVersionUID = 1L;
}
```

[4] The reason for the 31 message field limit is due to how eBus serializes messages. Go here to learn more.

Step 2: Message Annotation

`@EFieldInfo` run-time, class-level attribute defines message field serialization order. This required attribute is necessary because eBus cannot depend on the Java API to provide a consistent class field order. This field also defines the de-serialization constructor parameter ordering.

`@EReplyinfo` run-time, class-level attribute is required for *request* messages only. This annotation defines the `EReplyMessage`-derived message classes which may be sent in reply to this request. A replier is limited to sending a reply message that is listed in `CatalogOrder`'s `@EReplyInfo` annotation *and its super class' annotation*. Since all requests must descend from `ERequestMessage` which has the annotation `@EReplyInfo(replyMessageClasses={EReplyMessage.class})`, the class `EReplyMessage` may be sent in reply to any request.

An attempt to send a reply message unsupported by the request results in a *run-time* exception. This error is not found at compile time.

```
import java.io.Serializable;
import net.sf.eBus.messages.ERequestMessage;

@EFieldInfo(fields={"orderId", "price", "stockQty"})
@EReplyInfo(replyMessageClasses={CatalogOrderReply.class})
public final class CatalogOrder
    extends ERequestMessage
    implements Serializable
{
     See step 3.

    public final String orderId;
    public final Money price;  See step 5.
    public final int quantity;
    private static final long serialVersionUID = 1L;
}
```

Step 3: De-serialization constructor

Every eBus message subclass must provide a "de-serialization" constructor which first takes the required superclass arguments and then its own field types as arguments. The parameter ordering is defined by the `@EFieldInfo` class-level annotation.

`@EFieldInfo` fields are cumulative from the root class to base class. Given the following class hierarchy:

1. Class `CatalogOrderReply` extends `EReplyMessage` and has four fields: `subPrice`, `quantity`, `shippingCost`, and `totalPrice`.
2. Class `EReplyMessage` extends `EMessage` and has two fields: `replyStatus` and `replyReason`.
3. Class `EMessage` is the message root class and has two fields: `subject` and `timestamp`.

Class `CatalogOrderReply` has eight fields total: `subject`, `timestamp`, `replyStatus`, `replyReason`, `subPrice`, `quantity`, `shippingCost`, and `totalPrice`. This means that the `CatalogOrderReply` de-serialization constructor must have the same eight fields in the same order.

```
import java.io.Serializable;
import net.sf.eBus.messages.ERequestMessage;

@EFieldInfo (fields={“orderId”, "price", "quantity"})
@EReplyInfo (replyMessageClasses={CatalogOrderReply.class})
public final class CatalogOrder
    extends ERequestMessage
    implements Serializable
{
    // subject and timestamp are EMessage fields and must be the first two
    // constructor arguments.
    public CatalogOrder(final String subject,
                        final long timestamp,
                        final String orderId,
                        final Money price,
                        final int quantity) {
        super (subject, timestamp);

        this.orderId = orderId;
        this.price = price;
        this.quantity = quantity;
    }

    public final String orderId;
    public final Money price;  See step 5.
    public final int quantity;
    private static final long serialVersionUID = 1L;
}
```

Step 4: Application constructors

All other constructors are used to create new messages for transmission to recipients. These constructors may call the de-serialization constructor so that argument checking and data field assignment is done in one location.

```
import net.sf.eBus.messages.ERequestMessage;
import java.io.Serializable;

@EFieldInfo (fields={"orderId", "price", "stockQty"})
@EReplyInfo (replyMessageClasses={CatalogOrderReply.class})
public final class CatalogOrder
    extends ERequestMessage
    implements Serializable {

    // subject and timestamp are ENotificationMessage fields and must be the first two
    // constructor arguments.
    public CatalogOrder(final String subject,
                        final long timestamp,
                        final String orderId,
                        final Money price,
                        final int quantity) {
        super (subject, timestamp);

        this.orderId = orderId;
        this.price = price;
        this.quantity = quantity;
    }

    // This constructor requires the message subject and message fields, and generates
    // the message timestamp.
    public CatalogOrder(final String subject,
                        final String orderId,
                        final Money price,
                        final int quantity) {
        this (subject, System.currentTimeMillis(), orderId, price, quantity);
    }

    public final String orderId;
    public final Money price;  See step 5.
    public final int quantity;
    private static final long serialVersionUID = 1L;
}
```

Step 5: Message field type definition

eBus allows applications to define data types for message fields. This is done by extending the `EField` class and providing the required de-serialization constructor. The code below shows the `Money` data type definition. Like an eBus message, all fields must be `public final` and a supported eBus type.

eBus allows placing a subclass instance into an `EField` message field. For example, you define an abstract class `AbstractInfo` which extends `EField`. You then use `AbstractInfo` as a field in your message `CatalogUpdate`. You then define a concrete class `DetailInfo` extending `AbstractInfo`. eBus supports placing a `DetailInfo` instance into `CatalogUpdate` message. This technique allows you to update the message `CatalogUpdate` contents without changing the message class. All you need to do is create new `AbstractInfo` subclasses, each containing the desired information.

```
import java.io.Serializable;
import java.math.BigDecimal;
import net.sf.eBus.messages.EField;
import net.sf.eBus.messages.EFieldInfo;

@EFieldInfo (fields={"price", "currency", "qty"})
public final class Money
    extends EField
    implements Serializable {

    // The de-serialization constructor for creating a new Money instance.
    // Note: EField has no fields of its own so the default super () call works.
    public Money(final BigDecimal px, final Currency currency, final int qty) {
        super ();

        this.price = px;
        this.currency = currency;
        this.qty = qty;
    }

    public final BigDecimal price;
    public final Currency currency;
    public final int qty;
    private static final long serialVersionUID = 1L;
}
```

Step 6: Message class compilation

If a message class is repeatedly sent/received by an application, it is recommended that the message class be compiled by calling `EMessage.compile(Class<?>...)`. This static method generates code specifically for serializing/de-serializing the message class instances. This generated code is approximately twice as fast as the generic serialize/de-serialize code.

Message compilation also compiles all `EField` classes contained in the message class. Compilation cascades down to leaf fields which are pre-defined eBus field types.

```
try {
    // Speed up CatalogOrder serialization/de-serialization.
    EMessage.compile(CatalogOrder.class);
} catch (IllegalArgumentException | InvalidMessageException jex) {
    // Message compilation failed.
}
```

Note: as of eBus v. 4.4.0, all messages are automatically compiled. `EMessage.compile` is no longer in available in v. 4.4.0 or later. If this automatic compilation occurs at an inopportune time, then it can be forced by calling `net.sf.eBus.messages.DataType.findType(`*`message`*`.class)`. This will trigger the automatic compilation of *`message`*`.class`.

Step 7: The reply message.

Shown below is the `CatalogOrderReply` message referenced in the `CatalogOrder @EReplyInfo` annotation and in step 3.

```
import java.io.Serializable;
import net.sf.eBus.messages.EReplyMessage;
import static net.sf.eBus.messages.EReplyMessage.ReplyStatus;

@EFieldInfo (fields={“orderId”, "price", "stockQty"})
public final class CatalogOrderReply extends EReplyMessage implements Serializable {
    // subject and timestamp are ENotificationMessage fields and must be the first two
    // constructor arguments. replyStatus and replyReason are EReplyMessage fields and
    // must be the next two fields
    public CatalogOrderReply(final String subject,
                             final long timestamp,
                             final ReplyStatus replyStatus,
                             final String replyReason,
                             final Money subPrice,
                             final int quantity,
                             final Money shippingCost,
                             final Money totalPrice) {
        super (subject, timestamp, replyStatus, replyReason);

        this.price = price;
        this.quantity = quantity;
        this.shippingCost = shippingCost;
        this.totalPrice = totalPrice;
    }

    // This constructor requires the message subject and message fields, generating
    // the message timestamp, reply status, and reply reason.
    public CatalogOrderReply(final String subject,
                             final Money subPrice,
                             final int quantity,
                             final Money shippingCost,
                             final Money totalPrice) {
        this (subject, System.currentTimeMillis(),
              ReplyStatus.OK, null,
              subPrice, quantity, shippingCost, totalPrice);
    }

    public final Money subPrice;
    public final int quantity;
    public final Money shippingCost;
    public final Money totalPrice;
    private static final long serialVersionUID = 1L;
}
```

Arrays and List Fields

eBus field arrays are homogenous *except* for `File[]`, `BigDecimal[]`, `BigInteger[]`, `InetAddress[]`, `InetSocketAddress[]`, `EField[]`, and `EMessage[]` arrays. The reason the are heterogenous is because the array types are not marked `final`. That means an array slot may contain the array type subclass instance. This is not a problem for `BigDecimal` and `InetAddress` since eBus uses a static `valueOf` method to de-serialize to the correct target object. But for `File`, `BigInteger`, and `InetSocketAddress` array types, eBus assumes that the serialized field is of the specified array type. If the original message field contained a subclass instance, then that instance will *not* be de-serialized correctly since the parent class instance will be created and not the subclass.[5]

This is not the case with `EField` and `EMessage` arrays. For `EField` arrays, eBus serializes the field class together with the field instance. For `EMessage` arrays, eBus serializes the message key together with the message instance. This allows eBus to de-serialize the correct message subclass.

The upside to field and message arrays is that they allow for heterogenous messages to be stored in a single array. The downside is that such arrays serialize to large sizes (perhaps too large for eBus to handle) and are slow to de-serialize.

Another downside is that arrays are not immutable. It is possible to put a new value in an array slot post construction. This violates eBus requirement that a message instance is immutable.

The solution to this is `net.sf.eBus.messages.EFieldList` and `EMessageList` introduced in eBus 4.4.0. If an application has an `EField` subclass `Money`, then `EFieldList<Money>` can be used to transmit zero or more money instances. Likewise `EMessageList<CatalogOrder>` is used to transmit zero or more catalog order requests.

The downside is that all fields must be of the same class (`Money`) and all messages must have the same *message key* (the same message class and subject). The upside is that the serialized list takes up far less space and is faster to de-serialize.

`EFieldList` and `EMessageList` are also functionally immutable. The list contents can be modified up until the list is serialized or de-serialized.Once a field/message list is sent or received, any attempt to modify the list will result in a thrown `UnsupportedOperationException`.

[5] This also applies to a simple `File`, `BigInteger`, and `InetSocketAddress` field as well.

Connecting Up

eBus' purpose is to transmit messages between applications with minimal application development effort. The eBus interface and feed API is the same whether exchanging messages within a JVM or between JVMs ***except*** when using inter-JVM messaging, you need to use a `EFeed.FeedScope.LOCAL_AND_REMOTE` or `REMOTE_ONLY` scope. `LOCAL_ONLY` feed scope prevents a message from being sent to or received from a remote JVM.

An eBus application may open a service port which accepts client connections, open a client connection to a eBus service, or both. Whichever topology is chosen:

Only one connection is allowed between an eBus application pair regardless of which application initiated the connection.

A second connection is immediately rejected.

So let's see how to get eBus applications talking to each other.

Appendix A shows how eBus messages are serialized to the wire.

Appendix B shows how eBus protocol used between eBus applications.

Step 1: Opening an eBus service

An eBus service is as simple as calling `net.sf.eBus.client.EServer.openServer(port)` - if you want to use the default settings. Listed below are eBus server parameters. Keyword shows the matching eBus configuration file keyword used to set the parameter via the eBus configuration file (step 5).

Name: Service names list.
Description: Comma-separated list of service names. The service names are used to form the keys below.
Type: String
Optional? Yes.
Check: Does not contain empty names (",,"). Service names have no formatting restrictions.
Default: No services.
Keyword: `eBus.services`

Name: TCP port number.
Description: The TCP service port number.
Type: Integer
Optional? No.
Check: 0 ≤ port ≤ 65,536
Default: None.
Keyword: `eBus.service.`*`service`*`.port`

Name: Positive address filter.
Description: Specifies what Internet address or Internet address and TCP port pairs may connect to this service. This is a positive filter only. It is not a negative filter which specifies what addresses may not connect to the eBus service. If no address filter is provided then all connections are accepted. See step 4 for further information.
Type: `net.sf.eBus.client.AddressFilter`
Optional? Yes.
Check: None.
Default: No filter, all client connections are accepted.
Keyword: `eBus.service.`*`service`*`.addressFilter`

Name: Service selector thread name.
Description: The server socket is associated with this selector thread (see eBus network configuration, step 3).
Type: String
Optional? Yes.
Check: Selector name may not be `null`, empty, or unknown.
Default: Default selector specified by network configuration (step 3).
Keyword: `eBus.service.`*`service`*`.serviceSelector`

Name: Accepted connection selector thread name.
Description: Accepted client sockets are associated with this selector thread (see eBus network configuration, step 3).
Type: String
Optional? Yes.
Check: Selector name may not be `null`, empty, or unknown.
Default: Default selector specified by network configuration (step 3).
Keyword: `eBus.service.`*`service`*`.connectionSelector`

Name: Input buffer size.

Description: Set each accepted socket input buffer size to this number of bytes. The more data you expect to receive on the socket per second, the larger you should set this value.
Type: Integer
Optional? Yes.
Check: > zero.
Default: 2,048 bytes.
Keyword: `eBus.service.`*`service`*`.inputBufferSize`

Name: Output buffer size.
Description: Set each accepted socket output buffer size to this number of bytes. The more data you expect to send on the accepted socket per second, the larger you should set this value.
Type: Integer
Optional? Yes.
Check: > zero.
Default: 2,048 bytes.
Keyword: `eBus.service.`*`service`*`.outputBufferSize`

Name: Buffer byte order
Description: Set the socket input and output buffer to this byte order.
Type: Integer
Optional? Yes.
Check: must be either `ByteOrder.BIG_ENDIAN` or `ByteOrder.LITTLE_ENDIAN`.
Default: `ByteOrder.LITTLE_ENDIAN`
Keyword: `eBus.service.`*`service`*`.byteOrder`

Name: Output message queue size.
Description: Set each accepted socket's maximum output message queue size to this value. When the output message queue reaches this size, then the accepted socket is automatically closed. If this value is set to zero, then the message queue is allowed to grow without limit.
Type: Integer
Optional? Yes.
Check: ≥ zero.
Default: Zero (unlimited queue size).
Keyword: `eBus.service.`*`service`*`.messageQueueSize`

Name: Heartbeat delay.
Description: Send heartbeats at this millisecond rate. If set to zero, heart-beating is not performed. The heartbeat delay is reset when a message is received.
Type: Long (milliseconds)
Optional? Yes.
Check: ≥ zero.
Default: Zero (no heart-beating)
Keyword: `eBus.service.`*`service`*`.heartbeatDelay`

Name: Heartbeat reply delay.
Description: Wait this many milliseconds for a reply to a heartbeat. If set to zero, then wait indefinitely for a heartbeat reply.
Type: Long (milliseconds)
Optional? Yes.
Check: ≥ zero.
Default: Zero (wait indefinitely)
Keyword: `eBus.service.`*`service`*`.heartbeatReplyDelay`

Example of how to open and close an eBus service programmatically:

```
import java.nio.ByteOrder;
import net.sf.eBus.client.EServer;

EServer.openServer(12345,  // TCP service port.
                   filter, // Allowed address filter.
                   8192,   // Accepted client socket input buffer is 8,192 bytes.
                   8192,   // Accepted client socket output buffer is 8,192 bytes.
                   ByteOrder.BIG_ENDIAN, // Use big-endian byte ordering.
                   100,    // Outbound message queue size is 100 pending messages.
                   "serverSelector", // Server socket uses this selector.
                   "clientSelector", // Accept sockets assigned to this selector.
                   30000L, // Send heartbeat every 30 seconds.
                   500L);  // Wait 500 milliseconds for a heartbeat reply.

EServer.closeServer(12345);
```

If all eBus services need to be closed at one time, then call:

```
EServer.closeAllServers();
```

Step 2: Opening an eBus client connection

Opening an eBus client connection is also simple if willing to accept the default settings: `net.sf.eBus.client.ERemoteApp(InetSocketAddress)`. The full client connection settings are listed below. Keyword shows the matching eBus configuration file keyword used to set the parameter via the eBus configuration file (step 5).

Name: Connection names list.
Description: Comma-separated list of connection names. The connection names are used to form the keys below.
Type: String
Optional? Yes.
Check: Does not contain empty names (",,"). Connection names have no formatting restrictions.
Default: No connections.
Keyword: `eBus.connections`

Name: Internet address and port.
Description: Internet address and port of the remote eBus service.
Type: `InetSocketAddress`
Optional? No.
Check: Not `null`.
Default: No default.
Keyword: `eBus.connection.`*`connection`*`.host` and `eBus.connection.`*`connection`*`.port`

Name: Bind port.
Description: Bind the local port to this value when connecting. This may be required if the remote eBus uses an address filter with a specified port.
Type: Integer.
Optional? Yes.
Check: ≥ zero.
Default: `AsyncSocket.ANY_PORT` (local port assigned by the OS).
Keyword: `eBus.connection.`*`connection`*`.bindPort`

Name: Selector thread name.
Description: Client socket is associated with this selector thread (see eBus network configuration, step 3).
Type: String
Optional? Yes.
Check: Selector name may not be `null`, empty, or unknown.
Default: Default selector specified by network configuration (step 3).
Keyword: `eBus.connection.`*`connection`*`.selector`

Name: Input buffer size.
Description: Set the socket's input buffer size to this number of bytes. The more data you expect to receive on the socket per second, the larger you should set this value.
Type: Integer
Optional? Yes.
Check: > zero.
Default: 2,048 bytes.
Keyword: `eBus.connection.`*`connection`*`.inputBufferSize`

Name: Output buffer size.
Description: Set the socket's output buffer size to this number of bytes. The more data you expect to send on the accepted socket per second, the larger you should set this value.
Type: Integer
Optional? Yes.
Check: > zero.
Default: 2,048 bytes.
Keyword: `eBus.connection.`*`connection`*`.outputBufferSize`

Name: Buffer byte order.
Description: Set the socket input and output buffer to this byte order.
Type: String
Optional? Yes.
Check: must be either `ByteOrder.BIG_ENDIAN` or `ByteOrder.LITTLE_ENDIAN`.
Default: `ByteOrder.LITTLE_ENDIAN`
Keyword: `eBus.connection.`*`connection`*`.byteOrder`

Name: Output message queue size.
Description: Set the socket's maximum output message queue size to this value. When the output message queue reaches this size, then the socket is automatically closed. If this value is set to zero, then the message queue is allowed to grow without limit.
Type: Integer
Optional? Yes.
Check: ≥ zero.
Default: Zero (unlimited queue size).
Keyword: `eBus.connection.`*`connection`*`.messageQueueSize`

Name: Reconnect flag.
Description: If `true`, then eBus will automatically re-establish the connection if lost.
Type: Boolean
Optional? Yes.
Check: None.
Default: `false` (no reconnecting).
Keyword: `eBus.connection.`*`connection`*`.reconnect`

Name: Reconnect delay.
Description: If the reconnect flag is `true`, then wait this many milliseconds before attempting to reconnect. This value is ignored if the reconnect flag is `false`.
Type: Long (milliseconds)
Optional? Yes.
Check: > zero.
Default: No default.
Keyword: `eBus.connection.`*`connection`*`.reconnectDelay`

Name: Heartbeat delay.
Description: Send heartbeats at this millisecond rate. If set to zero, heartbeating is not performed.
Type: Long (milliseconds)
Optional? Yes.
Check: ≥ zero.
Default: Zero (no heart-beating).
Keyword: `eBus.connection.`*`connection`*`.heartbeatDelay`

Name: Heartbeat reply delay.
Description: Wait this many milliseconds for a reply to a heartbeat. If set to zero, then wait indefinitely for a heartbeat reply. The heartbeat delay is reset when a message is received.
Type: Long (milliseconds)
Optional? Yes.
Check: ≥ zero.
Default: Zero (wait indefinitely).
Keyword: `eBus.connection.`*`connection`*`.heartbeatReplyDelay`

Example of how to open and an eBus connection programmatically:

```
import java.net.InetSocketAddress;
import java.nio.ByteOrder;
import net.sf.eBus.client.ERemoteApp;

address = new InetSocketAddress("192.168.3.1", 10000);
ERemoteApp.openConnection(address, // The target host and port.
                          12312,   // Bind local port to this value.
                          65536,   // Set the input buffer size to 65,536 bytes.
                          32768,   // Set the output buffer size to 32,768 bytes.
                          ByteOrder.BIG_ENDIAN,
                                   // Input and output buffers use this byte ordering.
                          100,     // Set the message queue max size to 100 messages.
                          "clientSelector", // Socket assigned to this selector.
                          true,    // Reconnect automatically.
                          500L,    // Reconnect after 500 milliseconds.
                          30000L,  // Send a heartbeat every 30 seconds.
                          500L);   // Wait 500 milliseconds for a heartbeat reply.

ERemoteApp.closeConnection(address);
```

If all eBus connections must be closed at one time, then call:

```
ERemoteApp.closeAllConnections();
```

Step 3: eBus network configuration

eBus uses the Java NIO selector to determine when a TCP connection is ready to be read from or written to. eBus interacts with `java.nio.channels.Selector` from a `net.sf.eBus.net.SelectorThread`. An application may specify multiple selector threads with different latency performance depending on the connection requirements. There are three types of selector threads:

- **Blocking**: the selector thread blocks on `java.nio.channels.Selector.select()`.
- **Spinning**: the selector thread spins on `java.nio.channels.Selector.selectNow()`. Spinning is best done when the selector thread is pinned and owns a single core. eBus currently does not support pinning a selector thread to a core.
- **Spin+Park**: the selector thread alternates between spinning on `java.nio.channels.Selector.selectNow()` and parking (calling `java.util.concurrent.locks.LockSupport.parkNanos(nanoTime)`). This is a compromise between pure blocking and pure spinning. The spin limit and nanosecond park time are configurable.
- **Spin+Yield:** the selector thread alternates between spinning on selectNow() and yielding (calling LockSupport.park()). The spin limit is configurable.

Note: eBus configures selector threads at application start. Network configuration ***must*** be placed into a properties file and the java command line parameter set to point to that properties file:

```
-Dnet.sf.eBus.net.config.file=<properties file path>
```

Name: Selector thread names list.
Description: a comma-separated list of selector thread names. The names should be unique (no duplicates). All duplicates are logged and ignored. The selector thread name is used to retrieve the remaining keys.
Type: String
Optional? Yes.
Check: Does not contain empty names (",,"). Selector names have no formatting restrictions.
Default: A single blocking selector thread with `java.lang.Thread.NORM_PRIORITY`.
Keyword: `eBus.net.selectors`

Name: Selector thread type.
Description: Specifies the selector behavior type.
Type: String
Optional? No.
Check: Must be either `blocking`, `spinning`, `spin+park`, or `spin+yield` (case-insensitive).
Default: No default
Keyword: `eBus.net.selectors.`*`selector`*`.type`

Name: Default selector thread.
Description: If a socket channel is not assigned to a specific selector thread, it is assigned to this selector thread. If multiple selectors are designated as the default, then it cannot be determined which selector will be used as the default.
Type: Boolean
Optional? Yes.
Check: None.
Default: False (not the selector thread).
Keyword: `eBus.net.selectors.`*`selector`*`.isDefault`

Name: Thread priority
Description: The selector thread is assigned this Java thread priority value.

Type: Integer
Optional? Yes.
Check: Must be ≥ `java.lang.Thread.MIN_PRIORITY` and ≤ `java.lang.Thread.MAX_PRIORITY`.
Default: `java.lang.Thread.NORM_PRIORITY`
Keyword: `eBus.net.selectors.`*`selector`*`.priority`

Name: `spin+park` or `spin+yield` spin limit.
Description: If `.type` is `spin+park` or `spin+yield`, then this is the number of times the selector thread will call `Selector.selectNow()` before parking. When the park is finished, the selector thread resets its spin count to this spin limit and goes back to spinning. Ignored if `.type` is not either `spin+park` or `spin+yield`.
Type: Integer
Optional? Yes.
Check: > zero
Default: `net.sf.eBus.net.ENetConfigure.DEFAULT_SPIN_LIMIT`
Keyword: `eBus.net.selectors.`*`selector`*`.spinLimit`

Name: `spin+park` park time
Description: If `.type` is `spin+park`, then this is the selector thread nanosecond park time. Ignored if `.type` is not `spin+park`.
Type: Integer
Optional? Yes.
Check: > zero
Default: `net.sf.eBus.net.ENetConfigure.DEFAULT_PARK_TIME`
Keyword: `eBus.net.selectors.`*`selector`*`.parkTime`

```
#
# Because network configuration must be done at application start,
# ENetConfigure cannot be set programmatically. The configuration must be
# placed in a properties file and that file specified using the Java
# -D command line option:
#     java -Dnet.sf.eBus.net.config.file=<property file path>
#

# Two separate select threads are used: a spinning market data, and a
# spin+park selector for stock orders.
eBus.net.selectors=mdSelector,stockSelector

# Market data selector is low latency, so uses a spinning selector thread.
eBus.net.selector.mdSelector.type=spinning
eBus.net.selector.mdSelector.isDefault=false
eBus.net.selector.mdSelector.priority=9

# Order selector is low latency but not as critical as the market data.
# So this selector uses spin+park so as not to monopolize a CPU core.
eBus.net.selector.stockSelector.type=spin+park
eBus.net.selector.stockSelector.isDefault=false
eBus.net.selector.stockSelector.priority=6
eBus.net.selector.stockSelector.spinLimit=3000000
eBus.net.selector.stockSelector.parkTime=1000
```

Step 4: Address filter

An eBus service can be configured to accept connections from a specified series of hosts and, optionally, TCP ports on that host. Address filters can be created programmatically by constructing an `AddressFilter` instance or from a text description: `AddressFilter.parse(String description)`. An address filter can also be described in an eBus properties file via the `eBus.service.addressFilter` property.

When specifying an address filter using text (`AddressFilter.parse` or properties file), the format is:

```
address filter ::= <address> [ ',' <address>]*
address ::= (<host> | <IPv4 address> | <IPv6 address>) [ ':' <port>]
<host> ::= a host name which may be successfully de-referenced by
java.net.InetAddress.getByName(String host).
<IPv4 address> ::= a valid IPv4 dotted notation address.
<IPv6 address> ::= a valid IPv6 colon delimited address.
```

An example address filter property follows. This filter allows connections from hosts 192.168.3.2 and 192.168.3.3. In both instances, the remote port must be bound to 55001.

```
eBus.service.addressFilter=192.168.3.2:55001,192.168.3.3:55001
```

eBus address filtering is positive only. That means it specifies those addresses from which clients may connect. If the client is not in the filter, then eBus immediately closes the newly accepted connection. eBus does not support a negative filter which specifies addresses from which client may *not* connect.

Step 5: eBus configuration file

eBus service and client connections may also be opened automatically on application start by specifying those connections in a properties file and setting the eBus command line parameter to that properties file name:

```
-Dnet.sf.eBus.net.config.file=<properties file path>
```

This properties file would contain the eBus network configuration properties (if the default network configuration is not used). A Sample Java configuration file is shown below. It is a text file using the Java properties format.

```
# Comments begin with '#' or '!' and run to the end of the line.
    # The '#' or '!' does not have to be in the first column.

# This application makes two connections: one to a market data publisher and
# the other to a service which handles requests to buy or sell stocks.
eBus.connections=marketData, stockOrders

# The market data publisher connection. Requires large input but but a small
# output buffer. The message queue is of unlimited size. Reconnect 500
# milliseconds after connection loss.
eBus.connection.marketData.host=192.168.3.1
eBus.connection.marketData.port=10000
eBus.connection.marketData.inputBufferSize=262144
eBus.connection.marketData.outputBufferSize=8192
eBus.connection.marketData.byteOrder=BIG_ENDIAN
eBus.connection.marketData.messageQueueSize=0
eBus.connection.marketData.selector=mdSelector
eBus.connection.marketData.reconnect=true
eBus.connection.marketData.reconnectTime=500

# The stock order connection has fewer messages and so has smaller than
# default buffer sizes. However, it is more aggressive in reconnecting. Also,
# the stock order service expects us to bind the local port to a particular
# value. The message queue limit is kept deliberately small. If the stock
# requests cannot be sent right away, then cancel the request.
eBus.connection.stockOrders.host=StocksRUs.com
eBus.connection.stockOrders.port=10001
eBus.connection.stockOrders.bindPort=55000
eBus.connection.stockOrders.inputBufferSize=16384
eBus.connection.stockOrders.outputBufferSize=16384
eBus.connection.stockOrders.byteOrder=BIG_ENDIAN
eBus.connection.stockOrders.messageQueueSize=10;  # messages.
eBus.connection.stockOrders.selector=stockSelector
eBus.connection.stockOrders.reconnect=true
eBus.connection.stockOrders.reconnectTime=50
```

```
# This application also accepts eBus connections. Since it sends many
# messages but receives few, its uses a large output buffer and a small input
# buffer. Allow at most 10,000 messages to queue up. Heart beats every 30
# seconds to verify that clients are still alive and well.
# This server accepts only two connections: one from 192.168.3.2 and
# 192.168.3.3.
# Both connections must bind their local port to 55001.
eBus.service.port=20000
eBus.service.addressFilter=192.168.3.2:55001,192.168.3.3:55001
eBus.service.inputBufferSize=8192
eBus.service.outputBufferSize=262144
eBus.service.byteOrder=BIG_ENDIAN
eBus.service.messageQueueSize=10000
eBus.service.serviceSelector=stockSelector
eBus.service.connectionSelector=stockSelector
eBus.service.heartbeatDelay=30000
eBus.service.heartbeatReply=500

# Two separate select threads are used: a spinning market data, and a
# spin+park selector for stock orders.
eBus.net.selectors=mdSelector,stockSelector

# Market data selector is low latency, so uses a spinning selector thread.
eBus.net.selector.mdSelector.type=spinning
eBus.net.selector.mdSelector.isDefault=false
eBus.net.selector.mdSelector.priority=9

# Order selector is low latency but not as critical as the market data.
# So this selector uses spin+park so as not to monopolize a CPU core.
eBus.net.selector.stockSelector.type=spin+park
eBus.net.selector.stockSelector.isDefault=false
eBus.net.selector.stockSelector.priority=6
eBus.net.selector.stockSelector.spinLimit=3000000
eBus.net.selector.stockSelector.parkTime=1000
```

Step 6: Connection notification

An application can be notified when the eBus service accepts a new client connection or when a client connection's status changes by implementing `net.sf.eBus.client.ESubscriber` interface and subscribing to the following notification message keys:

Service updates: `net.sf.eBus.client.ServerMessage:"/eBus"`

Client updates: `net.sf.eBus.client.ConnectionMessage:"/eBus"`

Note: the subscription feed scope is local only.

`ServerMessage` contains two fields:

1. `InetSocketAddress remoteAddress`: this accepted client connection address. Client connections which do not pass the address filter are not reported.
2. `int serverPort`: the client connection was accepted on this server port.

`ConnectionMessage` contains four fields:

1. `InetSocketAddress remoteAddress`: the client connection's remote address.
2. `int serverPort`: if the client connect was accepted by an EServer, then this is the server port. If this connection was initiated by the application, then this value is set to zero.
3. `ConnectionState state`: the client connection state is either `ConnectionState.LOGGED_ON` or `ConnectionState.LOGGED_OFF`.
4. `String reason`: If `state` is `LOGGED_OFF` and the state change is due to an exception, then the exception message is stored in this field. May be `null`.

The following code is an example `ConnectionMessage` subscriber. `ServerMessage` subscription is similar.

```
import net.sf.eBus.client.ConnectionMessage;
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EFeedState;
import net.sf.eBus.client.ESubscriber;
import net.sf.eBus.client.ESubscribeFeed;
import net.sf.eBus.messages.ENotificationMessage;

public final class ConnectionWatcher
    implements ESubscriber
{
    private final int mPort;
    private ESubscribeFeed mConnectFeed;

    public ConnectionWatcher(final int port) {
        mPort = port;
        mConnectFeed = null;
    }
```

```
    @Override public void startup() {
        // Watch for connections associated with one EServer port.
        mConnectFeed =
            ESubscribeFeed.open(
                this,
                ConnectionMessage.MESSAGE_KEY,
                FeedScope.LOCAL_ONLY,
                (m) -> {
                    return (((ConnectionMessage) m).serverPort == mPort);
                );
        mConnectFeed.subscribe();
    }

    @Override public void shutdown() {
        mConnectFeed.close();
        mConnectFeed = null;
    }

    @Override public void feedStatus(final EFeedState feedState,
                                     final ESubscribeFeed feed) {
        System.out.format("%s feed is %s.%n", feed.key(), feedState);
    }

    @Override public void notify(final ENotificationMessage msg,
                                 final ESubscribeFeed feed) {
        final ConnectionMessage connMessage = (ConnectionMessage) msg;
        final InetSocketAddress address = connMsg.remoteAddress;

        System.out.format("Connection to %s, port %d is %s, reason: %s.%n",
                          address.getAddress(),
                          address.getPort(),
                          connMessage.state,
                          (connMessage.reason == null ?
                           "(none)" :
                           connMessage.reason));
    }
}
```

Dispatcher

eBus uses a Dispatcher to forward messages to client. While Dispatchers cannot be accessed by an application, an application can configure Dispatchers. In order to configure a Dispatcher, you need to know how Dispatchers work.

eBus wraps client callbacks in `Runnable` task instances. Each eBus client has an `Queue<Runnable>` containing the callback tasks for the client. So when a callback task is created, it is added to the client's task queue. When the client task queue is empty and no callback task is being run, then the client is idle. When a new callback task is added to an idle client, then the client becomes runnable.

When a client transitions from idle to runnable, then the *client* is added to its Dispatcher run queue. Each client is associated with one Dispatcher run queue. A Dispatcher has a run queue of runnable clients and one or more threads watch the run queue for runnable clients to arrive. One thread removes the client from the run queue and then starts executing the client's queue tasks. The client is now in the running state.

The queued client tasks are executed until either the task queue is empty or the client exhausts its run quantum. If a run quantum is used, then it is reset to the configured amount when the client transitions from idle to runnable or when the quantum is exhausted. When exhausted, the client transitions from running to runnable and put back on the LIFO run queue.

Note: eBus does *not* use a preempting run quantum. If a client callback goes into an infinite loop, then that callback will take over the Dispatcher thread. The Dispatcher thread only checks if the quantum is exceeded when the callback returns. Since the callback in this example never returns, the Dispatcher thread does not detect that the run quantum is exceeded.

Dispatchers are configured using the following parameters. Note that Dispatcher configuration *must* be done using a properties file and the `-Dnet.sf.eBus.config.file=<properties file path>` Java command line parameter.

eBus Clients Are Single-Threaded

An eBus object is associated with a single Dispatcher. An eBus object exists in one of three states: idle, runnable, and running. Only when in the runnable state does the eBus client appear on the Dispatcher run queue and then only once at any given time. When a Dispatcher thread removes the eBus object from the run queue, then it is the only Dispatcher thread to acquire that object at any given time. It also means that the eBus object is no longer on the run queue.

In summary, an eBus object is either:

1. Idle: Not on the run queue and not accessed by a Dispatcher thread.
2. Runnable: On the run queue and not accessed by a Dispatcher thread.
3. Running: Not on the run queue and accessed by a Dispatcher thread.

This means that at most one Dispatcher thread has access at any one moment. Therefore, the eBus call out to objects is effectively single-threaded. The good news is that if an application object is *not* accessed by non-eBus Dispatcher threads, then that object does not have to use synchronization to protect its data members.

If an application object *is* accessed by non-eBus Dispatcher threads (`java.util.Timer` for instance), then the object will need to protect its data members against multi-threaded updates.

Dispatcher now comes in four (!) flavors

Dispatcher now provides four different ways of polling the next available client from the run queue:

1. **Blocking.** Dispatcher thread blocks on `Queue.poll()` until a non-`null` client is returned. This technique is the most CPU-friendly but also the slowest since the thread will likely lose its core while blocked and must wait to come back on core to complete `Queue.poll()` call.

   The run queue is implemented as `java.util.concurrent.LinkedBlockingQueue`.

   This is the default Dispatcher polling type.

2. **Spinning.** Dispatcher thread continuously calls `Queue.poll()` until a non-`null` client is returned. Since `poll()` is implemented as non-blocking, this method the most CPU-unfriendly but the fastest since the thread will likely remain on core.[6]

   The run queue is implemented as `java.util.concurrent.ConcurrentLinkedQueue`.

3. **Spin+Park.** Dispatcher thread calls `Queue.poll()` until *either* a non-`null` client is returned or the spin limit is reached. If the spin limit is reached, then `LockSupport.park(long nanos)` is called. yielding the core to another thread for the specified time. When the park completes, the spin count is reset and the thread goes back to spinning. This method is more CPU-friendly than pure spinning and faster than blocking. This technique is open to wider performance variance than spinning.

   The run queue is implemented as `java.util.concurrent.ConcurrentLinkedQueue`.

4. **Spin+Yield.** Similar to spin+park except the thread parks indefinitely (`LockSupport.park()`). This method is even more CPU-friendly than spin+park but with greater performance variance.

   The run queue is implemented as `java.util.concurrent.ConcurrentLinkedQueue`.

Combining eBus Dispatcher and non-eBus Threads

Using non-eBus threads matter only when eBus clients run on those threads. When this occurs critical sections are introduced back into the eBus clients. If the non-eBus threads do not interact with eBus clients, then there are no critical sections to worry about. If it is necessary for an eBus client to interact with a non-eBus thread, there are two ways to handle the critical section:

1. **Synchronize.** This is the best known solution. Wrap the critical section inside a `synchronize` block or `Lock` being careful to make this synchronized region as small as possible. If the critical section can be handled by an atomic, so much the better. If the synchronization is un-contended for most of the time (99% or better?), then this solution adds little overhead to performance. This solution is preferred if there is a high level of interaction between non-eBus thread and eBus client. The problem is correctly identifying the critical sections and their minimal size.

2. **Dispatch.** When the non-eBus needs to interact with an eBus client, it can wrap the code up in a `Runnable` and post it to the client via `net.sf.eBus.client.EClient.dispatch(Runnable, EObject)`. This solution introduces a thread-handoff to every interaction which is not trivial. But if

---

[6] eBus and Java currently do not currently support setting thread affinity for a processor core. When using pure spin polling, it would be best if the Dispatcher thread were "pinned" to a core and that core isolated from the operating system.

there is occasional interaction between the non-eBus thread and eBus client, this solution is preferred because critical sections do not need to be identified and fits it with how the eBus client operates.

The following code demonstrates how a non-eBus thread can dispatch a callback to an eBus client `EClient.dispatch` and a lambda expression.

```
public final class ValueAddedPublisher implements EPublisher, ESubscriber {
    // EPublisher, ESubscriber interface implemented here.
    // The next method is used to handle a non-eBus event provided by a
    // non-eBus thread. This event is left to the reader's imagination.
    public void handleOutsideEvent(final NonEBusEvent event) {
        // Put event handling code here.
    }
}
```

The non-eBus thread `run` method passes information to the `ValueAddedPublisher` instance stored in the data member `mEventHandler` as follows:

```
public void run() {
    // Do thread work here.
    // Does ValueAddedPublisher need to know about something?
    if (event has occurred) {
        // Yes. Forward the information to the ValueAddedPublisher instance.
        final NonEBusEvent event = new NonEBusEvent(...);
        EClient.dispatch(() -> mEventHandler.handleOutsideEvent(event),
                         mEventHandler);
    }
}
```

Dispatcher Configuration

Name: Dispatcher names list.
Description: a comma-separated list of dispatcher names. The names should be unique (no duplicates). All duplicates are logged and ignored. The dispatcher name is used to retrieve the remaining keys.
Type: `String`
Optional? Yes.
Check: Does not contain empty names (",,"). Dispatcher names have no formatting restrictions.
Default: A single blocking dispatcher with `java.lang.Thread.NORM_PRIORITY` and no run quantum. Used for all eBus clients.
Keyword: `eBus.dispatchers`

Name: Dispatcher thread count.
Description: Number of threads assigned to this dispatcher.
Type: `Integer`
Optional? No.
Check: > zero.
Default: `net.sf.eBus.client.eConfigure.DEFAULT_NUMBER_THREADS`
Keyword: `eBus.dispatcher.`*`dispatcher`*`.numberThreads`

Name: Dispatcher run queue type.
Description: How dispatcher threads poll the run queue.
Type: `net.sf.eBus.client.ThreadType`
Optional? Yes.
Check: Must be either `blocking`, `spinning`, `spin+park`, or `spin+yield` (case-insensitive).
Default: `net.sf.eBus.client.ThreadType.BLOCKING`
Keyword: `eBus.dispatcher.`*`dispatcher`*`.runQueueType`

Name: Dispatcher spin limit.
Description: The number of times a dispatcher thread calls `Queue.poll()` before parking or yield.
Type: `Integer`
Optional? Required if `.runQueueType` is set to `spin+park` or `spin+yield`. Ignored otherwise.
Check: > zero.
Keyword: `eBus.dispatcher.`*`dispatcher`*`.spinLimit`

Name: Dispatcher park time.
Description: Nanosecond park time after dispatcher thread reaches spin limit.
Type: Integer
Optional? Required if `.runQueueType` is set to `spin+park`. Ignored otherwise.
Check: > zero.
Keyword: `eBus.dispatcher.`*`dispatcher`*`.parkTime`

Name: Dispatcher thread priority.
Description: The priority given to each of the dispatcher threads.
Type: `Integer`
Optional? Yes.
Check: Must be ≥ `java.lang.Thread.MIN_PRIORITY` and ≤ `java.lang.Thread.MAX_PRIORITY`.
Default: `java.lang.Thread.NORM_PRIORITY`
Keyword: `eBus.dispatcher.`*`dispatcher`*`.priority`

Name: Client run quantum
Description: Client task processing time quantum.
Type: `Integer` (nanoseconds)

Optional? Yes.
Check: > zero.
Default: `net.sf.eBus.client.eConfigure.DEFAULT_QUANTUM`
Keyword: `eBus.dispatcher.`*`dispatcher`*`.quantum`

Name: Default dispatcher flag.
Description: If `true`, then marks this as the default dispatcher for clients that are not explicitly assigned to a dispatcher.
Type: `Boolean`.
Optional? Yes.
Check: None.
Default: `false`
Keyword: `eBus.dispatcher.`*`dispatcher`*`.isDefault`

Name: Dispatcher client classes.
Description: Clients instantiated from the specified classes are assigned to this dispatcher.
Type: a comma-separated list of class names. Names should be unique.
Optional? No if `isDefault` property is `false`; otherwise is ignored.
Check: List length > zero and all named classes are known.
Default: None.
Keyword: `eBus.dispatcher.`*`dispatcher`*`.classes`

The following properties show how to configure eBus dispatchers.

```
# Two separate dispatchers are used: a spinning dispatcher for market data
# clients and a spin+park selector for all others.
eBus.dispatchers=mdDispatcher, defaultDispatcher

# Market data dispatch is low latency, so uses a spinning selector thread
# and 10 microsecond quantum.
eBus.dispatcher.mdDispatcher.threadCount=4
eBus.dispatcher.mdDispatcher.runQueueType=spinning
eBus.dispatcher.mdDispatcher.priority=7
eBus.dispatcher.mdDispatcher.quantum=10000
eBus.dispatcher.mdDispatcher.isDefault=false
eBus.dispatcher.mdDispatcher.classes=com.acme.trading.MDHandler

# Default dispatcher is low latency but not as critical as the market data.
# So this dispatcher uses spin+park and 100 microsecond quantum so as not to
# monopolize a CPU core.
eBus.dispatcher.defaultDispatcher.threadCount=8
eBus.dispatcher.defaultDispatcher.runQueueType=spin+park
eBus.dispatcher.defaultDispatcher.spinLimit=2500000
eBus.dispatcher.defaultDispatcher.parkTime=1000
eBus.dispatcher.defaultDispatcher.priority=4
eBus.dispatcher.defaultDispatcher.quantum=100000
eBus.dispatcher.defaultDispatcher.isDefault=true
```

Note: The above configuration assigns all `com.acme.trading.MDHandler` instances to the same Dispatcher, `mdDispatcher`. But not all `MDHandler`s are created equal - some are more important than

others, requiring higher priority processing. The above configuration is unable to distinguish between `MDHandler` instances.

Section "Gentlemen, start your objects" shows how individual application objects can be assigned to an eBus Dispatcher.

Gentlemen, start your objects

In Dispatcher, it was shown that eBus contacts application objects in a single-threaded fashion and that application objects do not require synchronization *as long as non-eBus threads do not access those objects.* But there is a catch. Consider the following value-added publisher ;

```
public final class ValueAddedPublisher implements EPublisher, ESubscriber {
    private final EMessageKey mPubKey;
    private final EMessageKey mSubKey;
    private EPublishFeed mPubFeed;
    private ESubscribeFeed mSubFeed;

    public ValueAddedPublisher(final EMessageKey pubKey,
                               final EMessageKey subKey) {
        mPubKey = pubKey;
        mSubKey = subKey;
    }

    public void start() {
        mSubFeed = ESubscribeFeed.open(this, mSubKey, FeedScope.LOCAL, null);
        mSubFeed.subscribe();

        // Race condition between feedStatus callback and the next two lines.
        mPubFeed = EPublishFeed.open(this, mPubKey, FeedScope.LOCAL);
        mPubFeed.advertise();
    }

    @Override public void publishStatus(final EFeedState feedState,
                                        final EFeed pubFeed) {
        // Application-specific code here.
    }

    @Override public void feedStatus(final EFeedState feedState,
                                     final ESubscribeFeed feed) {
        // If the subscription feed is up, the publish feed is up.
        // If the subscription feed is down, the publish feed is down.
        // BUT IS mPubFeed OPEN AND ADVERTISED YET?
        // Perhaps not. In that case, the following code will fail.
        mPubFeed.updateFeedState(feedState);
    }

    @Override public void notify(final ENotificationMessage msg,
                                 final ESubscribeFeed feed) {
        // Application-specific code here.
    }
}
```

The above class is created and started by the application main thread:

```
public static void main(final String[] args) {
    final EMessageKey pubKey = new EMessageKey(OutputMessage.class, args[0]);
    final EMessageKey subKey = new EMessageKey(InputMessage.class, args[0]);
    final ValueAddedPublisher vaPublisher =
        new ValueAddedPublisher(pubKey, subKey);

    // vaPublisher starting life on the application's main thread.
    // Note: start-up is synchronous.
    vaPublisher.start();

    // More application code. Will reach here after start-up completes.
}
```

The race condition is due to `vaPublisher` being started in the application main thread and `feedStatus` called out by an eBus Dispatcher thread. One way to resolve this is to `synchronize` the `start` and `feedStatus` methods. But this adds overhead to all `feedStatus` calls just to handle the one-time object start.

Another solution is to notice that by switching the publish feed to open and advertise before the subscription, the problem goes away. But what if `ValueAddedPublisher` opened more feeds and the interaction between those feeds was more complex? Depending on a correct start up sequence to get away from synchronization is not robust solution.

What if `ValueAddedPublisher` could be started on an eBus Dispatcher thread? That guarantees that `feedStatus` will be called only after the object start completed. eBus v. 4.3.0 provides a mechanism to do just that.

A new interface `net.sf.eBus.client.EObject` was introduced which contains two default method declarations:

```
default void startup() {}
default void shutdown() {}
```

Interfaces `EPublisher`, `ESubscriber`, `EReplier`, and `ERequestor` all extend `EObject`. By changing the `start` method to:

```
@Override public void startup()
```

`vaPublisher` can now be started on the Dispatcher thread by registering it with eBus and then having eBus start up `vaPublisher`:

```
public static void main(final String[] args) {
    final EMessageKey pubKey = new EMessageKey(OutputMessage.class, args[0]);
    final EMessageKey subKey = new EMessageKey(InputMessage.class, args[0]);
    final ValueAddedPublisher vaPublisher =
        new ValueAddedPublisher(pubKey, subKey);

    // vaPublisher starting life on a Dispatcher thread.
    // Note: start-up is now asynchronous.
    EFeed.register(vaPublisher);
    EFeed.startup(vaPublisher);

    // More application code. Will reach here before start-up completes.
}
```

As the comments note, the `vaPublisher` start-up is switched from synchronous to asynchronous. It is up to the application to coordinate object start-up with the main thread if the main thread cannot proceed until object start-up completes.

The above code shows the application shows `main` starting up a single object, `vaPublisher`. `EFeed` has two other start-up methods:

```
public static void startup(final Set<EObject> clients)
public static void startupAll()
```

The first is useful when starting up a batch of specific application objects. This could be used when implementing a multi-tier start-up. For example, you have one specific object whose feeds must all be `EFeedState.UP` before starting the second object tier. The first tier object is started using `EFeed.startup(EObject)`. When that first tier object detects that all its feeds are up, then it calls `EFeed.startup(Set<EObject>)` on the second tier objects.

But a more common start-up technique is to create and register all the application objects[7] and then call `EFeed.startupAll()`. This method starts up all registered application objects that were not previously started or implicitly registered.[8] With this technique, the application does not need to gather up all the registered objects into a `Set` and explicitly start up those objects. Less bookkeeping is easier.

eBus also supports object shutdown with the following `EFeed` methods:

```
public static void shutdown(final EObject client)
public static void shutdown(final Set<EObject> clients)
public static void shutdownAll()
```

Like object start-up, these methods call `shutdown()` from object's Dispatcher thread. The first two method shutdown specific objects. The third method shuts down all previously started objects. When an object's `shutdown` completes, that object may be started again without registering the object again.

Shutting down an object does *not* de-register the object from eBus. Once an object is registered with eBus, it remains registered until the object is finalized.

`EFeed.shutdownAll()` is the easier way to shutdown all registered application objects when terminating the application, if a clean termination is required for a distributed system.

---

[7] Be sure the application maintains a strong reference to those registered objects or they will be GC'd.

[8] An application object is implicitly registered when it opens a feed.

Pinning Application Objects to a Dispatcher

The Dispatcher section mentions that an eBus properties configuration does not support object-level Dispatcher association, only classes can be associated with a Dispatcher. With eBus v. 4.3.0, `EFeed.register()` can be used to dynamically assign an application object with a Dispatcher. There are two `EFeed` registration methods which support this:

```
public static void register(final EObject object,
                            final String dispatcherName)

public static void register(final EObject object,
                            final String dispatcherName,
                            final Runnable startCb,
                            final Runnable shutdownCb)
```

In both cases, `object` is associated with the eBus Dispatcher named `dispatcherName`. The eBus configuration must contain a Dispatcher with that name, otherwise an `IllegalArgumentException` is thrown.

The second method allows the caller to use lamba expressions to define `object`'s start-up and shutdown behavior.

Going back to the Dispatcher example configuration, a new Dispatcher configuration may be added:

```
eBus.dispatchers=priorityDispatcher, mdDispatcher, defaultDispatcher

# Priority market data handlers are posted to a single-thread Dispatcher
# and that thread is pinned to a core which is isolated from the OS.
eBus.dispatcher.priorityDispatcher.threadCount=1
eBus.dispatcher.priorityDispatcher.priority=10
eBus.dispatcher.priorityDispatcher.quantum=10000
eBus.dispatcher.priorityDispatcher.isDefault=false
eBus.dispatcher.priorityDispatcher.classes=
```

Now `MDHandler` instances identified as high priority can be registered to "`priorityDispatcher`".

Registration Gotchas

An application object can only be registered with eBus *if it is not already registered*. Otherwise, an `IllegalStateException` is thrown.

The problem is that an application object is implicitly registered with eBus when it opens a feed. The application object must be registered before opening any feeds. But this will be the case when registering the object prior to calling `EFeed.startup` and the feeds are opened in the `startup()` method.

Note: when objects are implicitly registered, `startup()` is *not* called.

Another problem is that once an object is posted to a Dispatcher, that posting cannot change. So it is no possible to switch the object between Dispatchers during the object's lifetime. The only solution is to shutdown the object and create a replacement object which is then pinned to another Dispatcher.

As noted before, eBus maintains only a weak-reference to application objects. So when calling `EFeed.register(EObject object)`, be sure the application keeps a strong reference to `object`. Failure to do so will guarantee `object` is finalized in the next garbage collection.

State of the Union: eBus and SMC

A call stack provides the context for synchronous communication. A new stack frame is pushed on top of a call stack when calling a subroutine and the subroutine arguments are written into the stack frame. On returning from the subroutine, the top stack frame is popped off the stack and the return value copied to the new top frame. This returns the software back to the context of the subroutine call, the context needed to interpret the returned value.

But when sending an eBus request, the call stack that existed when the request was made is irretrievably gone when the asynchronous reply is delivered, taking its context with it. So how can context be maintained in the face of asynchronous messaging?

The industry standard solution is to use a finite state machine for asynchronous context. The current state defines how an object responds to a transition where the transition contains the message. This section shows how to define and integrate a finite state machine into an object and how to translate eBus callbacks into a state machine transition.

SMC: the State Machine Compiler takes a state machine definition and generates code in a target which implements that state machine. This section defines a state machine for an object that subscribes to a data point feed and publishes the moving average calculated from those data points.

Note: this section does not describe the SMC syntax or compiler settings used to generate the Java code which implements the state machine. This is covered thoroughly in the SMC Programmer's Manual available at the SMC website.

Demo Architecture

This demonstration consists of three components:

- **Source**: publishes integer data points in the [min, max) range at a configurable millisecond rate. Data point publishing is triggered using `net.sf.eBusx.util.Timer`.
- **Calculator**: subscribes to the data point feed and publishes the calculated moving average. The moving average size is configurable. Uses a finite state machine to decide when to publish a moving average message.
- **Sink**: subscribes to the moving average feed and outputs the received messages to the console.

This section presents the calculator finite state machine and shows how to integrate that state machine into the `Calculator` class and into eBus.

Calculator State Machine

There are five states in the `Calculator` FSM:

1. `Initializing`: This is the FSM start state. When entered, the data point subscription and moving average publishing feeds are opened. `Calculator` waits in this state for the data point feed to come up.
2. `FeedDown`: If the data point feed goes down after coming up, then the FSM goes immediately to this state from whatever state it is currently in. Upon entry, the moving average feed is marked as down and all moving average calculations are cleared. `Calculator` waits in this state for the data feed to come back up.
3. `Collecting`: Wait in this state, collecting data points until enough are collected to calculate the first moving average. When this happens move to the `Publishing` state.
4. `Publishing`: Publishes a new moving average each time a data point is received. Upon entry, the moving average feed is marked up and the first moving average notification message is published.
5. `Shutdown`: Enter this state when the demo is shut down.

There is a sixth "state" named `Default`. This is not a real state but is used to define a transition's default behavior if not explicitly defined in a state. One example is the `shutdown` transition. It is not defined in any state, so the `Default` state defines the global `shutdown` transition for all states. In this case, `shutdown` closes the data point subscription and moving average publishing feeds and goes to the `Shutdown` state.

What follows is the `Calculator` FSM definition which is stored in file `Calculator.sm`. Again, see the SMC Programmer's Manual for a detailed description of the SMC syntax.

```
%class Calculator       // Associates FSM with the Calculator class.
%package smcdemo
%fsmclass CalculatorFSM // FSM defined in class CalculatorFSM
%fsmfile CalculatorFSM  // FSM class stored in file CalculatorFSM.java
%access package // Actions defined in Calculator class with package-private access.

%import net.sf.eBus.client.EFeedState
%import net.sf.eBus.client.ESubscribeFeed
%import net.sf.eBus.messages.ENotificationMessage

%start CalculatorMap::Initializing // Defines FSM start state.

%map CalculatorMap
%%

// When the system starts, put the eBus feeds in place.
Initializing Entry {startFeeds();} {
    // When the feed state comes up, start collecting data points.
    // Note: transition signature matches
    // ESubscriber.feedStatus(EFeedState, ESubscribeFeed)
    feedState(feedState: final EFeedState, feed: final ESubscribeFeed)
      [feedState == EFeedState.UP]
        Collecting {}

    // Wait here for the feed to come up. nil is an internal loopback transition.
    feedState(feedState: final EFeedState, feed: final ESubscribeFeed)
        nil {}
}
// Wait here for the feed to come up.
```

```
FeedDown Entry {setPubState(EFeedState.DOWN); clearStats();} {
    // When the feed state comes up, start collecting data points.
    feedState(feedState: final EFeedState, feed: final ESubscribeFeed)
      [feedState == EFeedState.UP]
        Collecting {}

    // Wait here for the feed to come up.
    feedState(feedState: final EFeedState, feed: final ESubscribeFeed)
        nil {}
}

// Wait here until enough data points are in hand to publish the moving average.
Collecting {
    // When the required number of data points are collected, then start publishing.
    // Note: transition signature matches
    // ESubscriber.notify(ENotificationMessage, ESubscribeFeed)
    data(msg : final ENotificationMessage, feed: final ESubscribeFeed)
      [ctxt.addData(((DataPoint) msg).data) == true]
        Publishing {}

    // Continue collecting.
    data(msg : final ENotificationMessage, feed: final ESubscribeFeed)
        nil {}
}

// Publish the updated moving average every time a new data point is received.
Publishing Entry {setPubState(EFeedState.UP); publish();} {
    data(msg : final ENotificationMessage, feed: final ESubscribeFeed)
        nil
        {addData(((DataPoint) msg).data); publish();}
}

// The system is shutting down.
Shutdown {
    // Stay here forever.
    Default nil {}
}

// Default transitions.
Default {
    // When the feed state goes down, go immediately to FeedDown.
    // Do not pass go. Do not collect $200.
    feedState(feedState: final EFeedState, feed: final ESubscribeFeed)
      [feedState == EFeedState.DOWN]
        FeedDown {}

    // Otherwise, ignore the feed state going up when it is already up.
    feedState(feedState: final EFeedState, feed: final ESubscribeFeed)
        nil {}

    // Ignore unexpected data points.
    data(msg : final ENotificationMessage, feed: final ESubscribeFeed)
        nil {}

    shutdown()
        Shutdown
        {stopFeeds();}
}

%% // end of CalculatorMap
```

Integrating the FSM into Calculator

The SMC-generated finite state machine is integrated into the `Calculator` class by creating a data member to store the FSM instance, instantiating the FSM, and defining the FSM action methods:

```
public final class Calculator
    implements ESubscriber, EPublisher {

    // FSM class name matches %fsmClass CalculatorFSM in SMC definition.
    private final CalculatorFSM mFsm;

    public Calculator(final int size, final String subject) {
        // Pass this calculator reference to the FSM which allows the FSM to
        // call Calculator methods.
        mFsm = new CalculatorFSM(this);
        ...
    }

    // The following methods are accessed by the FSM in the Entry and
    // transition action bodies.
    // Methods have package-private access to match "%access package" in
    // Calculator.sm.

    /* package */ void startFeeds() { ... }

    /* package */ void stopFeeds() { ... }

    /* package */ void setPubState(final EFeedState feedState) { ... }

    /* package */ void clearStats() { ... }

    /* package */ boolean addData(final int dataPoint) { ... }

    /* package */ void publish() { ... }
}
```

Integrating the FSM into eBus

This integration occurs when the `Calculator` instance is registered with eBus and when the data point subscription and moving average publishing feeds are opened.

```
/* package */ void register() {
    // Enter into the FSM's start state on start-up and issue a shutdown
    // transition on shut down.
    // Self registration is done because main cannot access mFsm in order to
    // set the start-up and shutdown callbacks.
    EFeed.register(this,
                   EFeed.defaultDispatcher(),
                   mFsm::enterStartState, // SMC-generated method.
                   mFsm::shutdown);
}

/* package */ void startFeeds() {
    mSubFeed = ESubscribeFeed.open(this,
                                   mSubKey,
                                   EFeed.FeedScope.LOCAL_ONLY,
                                   null);

    // Subscribe feed updates are converted directly into FSM transitions.
    // These two lines integrate eBus into the FSM.
    // Note: the feedState and data transition signatures must match the ESubscriber
    // callback signatures.
    mSubFeed.statusCallback(mFsm::feedState);
    mSubFeed.notifyCallback(mFsm::data);

    mSubFeed.subscribe();

    mPubFeed = EPublishFeed.open(this,
                                 mPubKey,
                                 EFeed.FeedScope.LOCAL_ONLY);
    mPubFeed.advertise();
}

// Ignore publish status updates.
@Override public void publishStatus(final EFeedState feedState,
                                    final EPublishFeed feed)
{}
```

Now eBus callbacks to the `Calculator` instance will be routed directly to FSM transitions and the FSM defines the `Calculator` response to that callback.

I. The combination of SMC and Lambda expressions makes it simple to integrate an FSM into your class and eBus. This technology is useful when defining a class with complex behavior, behavior that is beyond the capability of an enum and switch statements.

The complete code demonstrating how an SMC-generated finite state machine can be integrated into eBus is available from http://sourceforge.net/projects/ebus and stored in the release Utilities folder under `SmcDemoSrc_x_y_z.tgz` where `x_y_z` is the eBus version.

Appendices

Appendix A: Binary message layout

eBus external connections use binary serialization. Serialized eBus messages consist of a header and body. The body consists of fields whose number and type are listed in an `@EFieldInfo` class-level annotation. The header consists of the following:

Start Byte	End Byte	Length	Description
0	3	4	Total message length (includes these four bytes) **or** HEARTBEAT (0xC568) **or** HEARTBEAT_REPLY (0xE0C0) **Note:** Maximum message length is 32,767 bytes (including 16 byte header leaving 32,751 bytes for message).
4	7	4	Message key identifier. Uniquely identifies a message class, subject pair and is assigned by the *remote* application.
8	11	4	From feed identifier. Message is from this local message feed. Responses to this message are sent to this feed.
12	15	4	To feed identifier. This message is delivered to this remote message feed. Set to -1 if the local feed is not yet linked to a remote feed.
16	19	4	Message field mask (4-byte, signed integer). This is why messages are limited to 31 fields - one bit per field. If bit is not set, then field does not appear in the payload.
20	(length - 1)	(length - 20)	Serialized eBus message (EMessage). See below.

Only non-null fields are serialized. If a field is null, then its bit is set to zero in the message field mask. A non-null field has is message field mask bit set to one and then the field is serialized to the buffer.

The first 4 fields of 14 bytes is the message header. The last two fields are the serialized message body.

The following table describes how eBus serializes supported field types.

Type	Length	Description

`BigDecimal`	12	`Pos Sz Description`
		`0 8 BigDecimal.unscaledValue()`
		`8 4 BigDecimal.scale()`
		De-serialized using `BigDecimal.valueOf(long, int)`.

`BigInteger`	2 + byte array length	`Pos Sz Description`
		`0 2 byte[] length`
		`2 n byte[] value`

`boolean/Boolean`	1	`Pos Sz Description` `0 1 1 if true, zero if false.`
`byte/Byte`	1	`Pos Sz Description` `0 1 byte or Byte.byteValue()`
`char/Character`	2	`Pos Sz Description` `0 2 char or Character.charValue()`
`Class`	2 + class name length	Performs `String` serialization on `Class.getName()`. De-serialized using `Class.forName(String)`.

`Date`	8	`Pos Sz Description`
		`0 8 Date.getTime()`
		De-serialized using `new Date(long)`.

`double/Double`	8	`Pos Sz Description` `0 8 double or Double.doubleValue()`

`enum`	2	`Pos Sz Description`
		`0 2 enum.ordinal()`
		De-serialized using `enumclass.getEnumConstants()[ord]`.

`EField`	*n*	`Pos Sz Description`
		`0 4 field mask, signed int.`
		`4 n message fields`
		The 31 bit field mask is why `EField` classes are limited to 31 fields.

`File`	2 + file name length	Performs `String` serialization on `File.getPath()`. Path length limited to 1,024 characters. De-serialized using `new File(String)`.
`float/Float`	4	`Pos Sz Description` `0 4 float or Float.floatValue().`

Type	Length	Description

`InetAddress`	4 (IPv4)
	8 (IPv6)
		`Pos Sz Description`
		`0 2 address size (4 or 8 bytes)`
		`2 4 InetAddress.getBytes() (v4)`
		`2 8 InetAddress.getBytes() (v6)`
		De-serialized using
		`InetAddress.getByAddress(byte[]).`

`InetSocketAddress`	8 (IPv4)
	12 (IPv6)
		`Pos Sz Description`
		`0 6 IPv4 address`
		`6 4 TCP port OR`
		`0 10 IPv6 address`
		`10 4 TCP port`
		De-serialized using
		`new InetSocketAddress(addr, port).`

`int/Integer`	4	`Pos Sz Description` `0 4 int or Integer.intValue()`
`long/Long`	8	`Pos Sz Description` `0 8 long or Long.longValue()`

`EMessageKey`	*n*	`Pos Sz Description`
		`0 n Class serialize message class`
		`8 m String serialize subject`
		De-serialized using
		`new EMessageKey(Class, String).`

`short/Short`	2	`Pos Sz Description` `0 2 short or Short.shortValue()`

`String`	2 + string length	`Pos Sz Description`
		`0 2 String.length()`
		`2 n String.getBytes(UTF8 charset)`
		Note: strings limited to 1,024 characters.
		De-serialized using
		`new String(byte[], CharSet).`

`java.net.URI`	2 + URI string length	Performs `String` serialization on `URI.toString()`. De-serialized using `new URI(String).`
`java.time.Duration`	8	Serialized using `Duration.toNanos()`. De-serialized using `Duration.ofNanos()`.
`java.time.Instant`	8	Serialized using `Instant.toEpochMilli()`. De-serialized using `Instant.ofEpochMilli()`.
`java.time.LocalDate`	8	Serialized using `LocalDate.toEpochDay()`. De-serialized using `LocalDate.ofEpochDay()`.
`java.time.LocalTime`	8	Serialized using `LocalTime.toNanoOfDay()`. De-serialized using `LocalTime.ofNanoOfDay()`.

Type	Length	Description
java.time.LocalDateTime	16	Serialized using LocalDateTime.toLocalDate(), LocalDateTime.toLocalTime(). De-serialized using LocalDateTime.of(LocalDate, LocalTime).
java.time.MonthDay	8	Serialized using MonthDay.getMonthValue(), MonthDay.getDayOfMonth(). De-serialized using MonthDay.of(int, int).
java.time.OffsetTime	12	Serialized using OffsetTime.toLocalTime(), OffsetTime.getOffset(). De-serialized using OffsetTime.of(LocalTime, ZoneOffset)
java.time.OffsetDateTime	20	Serialized using OffsetDateTime.toLocalDate(), OffsetDateTime.toLocalTime(), OffsetDateTime.getOffset(). De-serialized using OffsetDateTime.of(LocalDate, LocalTime, ZoneOffset).
java.time.Period	12	Serialized using Period.getYears(), Period.getMonths(), Period.getDays(). De-serialized using Period.of(int, int, int).
java.time.YearMonth	8	Serialized using YearMonth.getYear(), YearMonth.getMonthValue(). De-serialized using YearMonth.of(int, int).
java.time.Year	4	Serialized using Year.getValue(). De-serialized using Year.of(int).
java.time.ZoneOffset	8	Serialized using ZoneOffset.getTotalSeconds(). De-serialized using ZoneOffset.ofTotalSeconds(int).
java.time.ZoneId	zone ID string length	Serialized using ZoneId.getId(). De-serialized using ZoneId.of(String).
java.time.ZonedDateTime	8 + zone ID string length	Serialized using ZonedDateTime.toLocalDate(), ZonedDateTime.toLocalTime(), ZonedDateTime.getZone(). De-serialized using ZonedDateTime.of(LocalDate, LocalTime, ZoneId.

Type	Length	Description

type`[]` array	*n*	`Pos Sz Description`
		`0  2 `*array*`.length`
		`2  `*n*` `*type*` serialize each element.`
		De-serialized using `Array.newInstance(Class, int)` and then using *type* to de-serialize each element.

Appendix B: eBus connection protocol

`net.sf.eBus.client.ERemoteApp` provides the interface between remote eBus applications. `ERemoteApp` represents remote publishers and repliers to the local JVM and local subscribers and requestors to the remote JVM. This section describes how `ERemoteApp` handles messages from the remote JVM and `EPublisher`, `ESubscriber`, `ERequestor`, and `EReplier` callbacks in the local JVM. First, a brief description of how eBus applications shake hands when connecting.

1. eBus client successfully connects to a remote eBus service.
2. eBus client sends a `net.sf.eBus.client.sysmessages.LogonMessage` to remote eBus. This system message contains the JVM identifier returned by `(ManagementFactory.getRuntimeMXBean()).getName()`.
3. eBus client waits for a `net.sf.eBus.client.sysmessages.LogonReply`. This message contains status and reason fields. If status is `ReplyStatus.OK`, then both sides of the connection go to step 4. If status is `ReplyStatus.ERROR`, then the remote eBus rejected the connection due to this being a redundant connection from the local JVM. The connection is then closed.
4. Both sides exchange `net.sf.eBus.client.sysmessages.AdMessage` for all active publisher and replier advertisements. This message contains four fields: message class, message subject, ad status, and ad type. The message class is sent as a `String` and not a `Class` field because the local JVM cannot be certain that remote JVM supports the named class. If `Class` was used, then the message de-serialization could fail when `Class.forName()` is called, resulting in a message discard. Sending the class name as a `String` avoids this problem. If the message class is not supported on the remote JVM, then the advertisement is ignored. The ad status field is set to either `AdStatus.ADD` or `AdStatus.REMOVE`. In this case the status is `ADD`. The ad type is a `net.sf.eBus.messages.EMessage.MessageType` and is either `NOTIFICATION` or `REPLY`.
5. When `ERemoteApp` advertisement transmission is complete, then a `net.sf.eBus.client.sysmessages.LogonComplete` is sent.

The following table describes how `ERemoteApp` responds to system messages:

LogonMessage

Fields:

`public final String eid`

Unique eBus identifier. Set to `java.lang.management.ManagementFactory.getRuntimeMXBean().getName()` which returns a unique JVM identifier.

Description:

An eBus application sends this as the first message after successfully connecting.

Response:

`net.sf.eBus.client.sysmessages.LogonResponse`

LogonResponse

Fields:

`public final String eid`

Unique eBus identifier. See `LogonMessage`.

`public final ReplyStatus logonStatus`

If logon request is accepted, set to `ReplyStatus.OK_FINAL`. If logon request is rejected, then set to `ReplyStatus.ERROR`.

`public final String reason`

If logon request is rejected, then text explaining why it was rejected is stored here.

Description:

Sent in response to a `LogonMessage`, informing the remote eBus application whether the request was accepted or rejected. If rejected, the connection will be closed. If accepted, advertisements and message key identifiers will be exchanged. A `LogonCompleteMesage` is sent to inform the other side that advertisement and message key transmission is complete.

Response:

```
net.sf.eBus.client.sysmessages.KeyMessage
net.sf.eBus.client.sysmessages.AdMessage
net.sf.eBus.client.sysmessage.LogonCompleteMessage
```

KeyMessage

Fields:

`public final int keyId`

Unique identifier assigned to message key. This is the value placed into the message header when transmitted.

`public final String keyClass`

Key's message class name. Sent as a `String` because this class may not be known to the receiving eBus application. If that is the case, then the receiving eBus application ignores this message.

`public final String keySubject`

Key's subject.

Description:

Provides the mapping between a 4-byte, signed integer and a message key. This integer identifier is used in a message header to identify the inbound message key which is used to de-serialize the message. This means that the message class name and subject do not need to be encoded as strings.

Response:

No response.

AdMessage

Fields:

`public final String messageClass`

Key's message class name.

`public final String messageSubject`

Key's subject.

`public final AdStatus adStatus`

Set to `AdMessage.AdStatus.ADD` when installing a new advertisement and `AdMessage.AdStatus.REMOVE` when retracting an existing advertisement.

`public final MessageType adMessageType`

Either `EMessage.MessageType.NOTIFICATION` for a publisher advertisement and `EMessage.MessageType.REQUEST` for a replier advertisement.

Description:

During the login process, used to announce publisher and replier advertisements with local & remote or remote scope to a remote application so it can be installed. If the remote application does not support messageClass, then the advertisement is ignored.

If `adStatus` is `AdStatus.ADD`, then opens either an `EPublishFeed` or `EReplyFeed` depending on whether the advertised message class is an `ENotificationMessage` or `ERequestMessage`.

If `adStatus` is `AdStatus.REMOVE`, then retracts the feed advertisement.

Response:
No response.

LogonCompleteMessage

Fields:
`public final String eid`
Unique eBus identifier. Set to `java.lang.management.ManagementFactory.getRuntimeMXBean().getName()` which returns a unique JVM identifier.

Description:
Denotes that the KeyMessage and AdMessage stream is completed. eBus subscriptions, notifications, requests, and replies may now be exchanged. Received advertisements are now processed (see below).

Response:
No response. Putting an advertisement in place may result in subscribe messages sent in return.

The following messages are sent after a successful login.

SubscribeMessage

Fields:
`public final String messageClass`
Key's message class name.

`public final String messageSubject`
Key's subject.

`public final EFeedState feedState`
Set to `EFeedState.UP` when subscribing and `EFeedState.DOWN` when retracting an existing subscription.

Description:
Used to subscribe or unsubscribe to the specified notification message key. If `feedState` is `EFeedState.UP`, then opens an `ESubscribeFeed`; otherwise unsubscribes the feed.

Response:
No response.

FeedStatusMessage

Fields:
`public final EFeedState feedState`
Set to `EFeedState.UP` or `EFeedState.DOWN`.

Description:
Used to inform local subscribers whether a remote notification feed is either up or down. This message does not contain message key fields because that is implied by the To Feed header field.

Response:
No response.

RemoteAck

Fields:
`public final int remaining`
The number of repliers for the given request.

Description:

When eBus receives a remote request and there are repliers to the request, then this message is sent before any replies to inform the remote eBus application about the number of repliers from this JVM.

Response:
No response.

CancelRequest

Fields:
No fields.

Description:
Cancels a remote request by canceling the referenced `ERequestFeed.ERequest`.

Response:
No response.

The following describes how application messages are handled, both inbound and outbound.

Appendix C: eBus protocol stack

This table describes the eBus binary protocol, its levels, and how configuration impacts this stack. eBus uses Java NIO to perform the socket I/O.

ERemoteApp

Description: Responsible for maintaining a connection to a remote eBus application.

`eBus.connection.`*`name`*`.host:` the remote application service is open on this host.

`eBus.connection.`*`name`*`.port:` the remote application service is open on this port.

`eBus.connection.`*`name`*`.bindPort:` bind the connection's local side to this port.

Input: Forwards messages to target `EFeed` instance, *except* system messages. See Appendix B for further information on how inbound system messages are handled.

Output: Passes outbound system and user messages to the associated `ETCPConnection` instance. If the `ETCPConnection` output message queue overflows, then the connection is closed and all queued messages are discarded. If the connection is set to reconnect, then the reconnect timer is set. (Note: system messages are not used to calculate queue depth.)

`eBus.connection.`*`name`*`.reconnect:` if `true`, then a lost connection is re-established. The default value is `false`.

`eBus.connection.`*`name`*`.reconnectTime:` specifies the millisecond rate at which reconnect attempts are made. The default value is 5 *seconds*. The setting is ignored if **`reconnect`** is set to `false`.

ETCPConnection

Description: Responsible for serializing outbound messages and deserializing inbound messages. Also responsible for queuing up outbound messages when the socket TCP window narrows. Sends the enqueued messages when the buffer overflow condition clears. Again, system messages are *not* counted against the outbound message queue length.

Input: De-serializes eBus messages directly from the `AsyncSocket` input buffer. Posts de-serialized messages to `ERemoteApp`.

The connection does not its own input buffer but uses the `AsyncSocket` input buffer which is configurable.

`eBus.connection.`*`name`*`.heartbeatReplyDelay:` if > zero, then wait this many milliseconds for a reply to a heartbeat. This timer is reset every time data is received from the far-end. This setting is ignored if **`heartbeatDelay`** is not set.

Output: Serializes outbound message directly to the socket output buffer using the `BufferWriter` interface which throws a `BufferOverflowException` if the socket output buffer overflows. This exception is caught by `ETCPConnection` and the message is posted to the message queue. When the socket output buffer is no longer full, forwards the queued messages until the queue is either empty or the socket output buffer is again full.

The connection does *not* have its own output buffer but uses the `AsyncSocket` output buffer which is configurable.

`eBus.connection.`*`name`*`.messageQueueSize:` if set and this size is exceeded, then the connection is immediately closed and the upstream `ERemoteApp` notified.

eBus.connection.*name*.heartbeatDelay: If > zero, then send a heartbeat to remote end after this many milliseconds of inactivity. That is, this timer is reset every time data is received from the far-end.

AsyncSocket

Description: Interface between `ETCPConnection` and the `SelectorThread`. Encapsulates the `SelectableChannel`, and input, output `ByteBuffer`s. Both are direct allocations.

Input: Passes input `ByteBuffer` directly to `ETCPConnection`.

eBus.connection.*name*.inputBufferSize: specifies the socket input buffer fixed size. Defaults to 2,048 bytes.

Output: Outbound messages are serialized to a socket output buffer via a `BufferWriter` instance. If the output buffer size is exceeded, then throws a `BufferOverflowException`.

eBus.connection.*name*.outputBufferSize: specifies the socket output buffer fixed size. Defaults to 2,048 bytes.

SelectorThread

Description: Watches `SelectableChannel` instances and performs the actual read, write and accept operations. Can be configured to block or spin when selecting.

eBus.connection.*name*.selector: specifies the selector thread used to monitor the socket channel.

Input: Reads bytes from a `SocketChannel` into the `AsyncSocket` input buffer.

Output: Passes the `AsyncSocket` output buffer to `SocketChannel.write(ByteBuffer)`.

Appendix D: Building eBus from source

These steps apply to eBus version 4.1.0 and later.

Requirements:

- Java 1.8 or better.
- Apache ant 1.9.7 or better.
- JUnit 4 or better.

Build steps:

1. Browse to https://www.sourceforge.net/projects/ebus .
2. Click on `Files` in the project menubar at the top of the page.
3. Select the desired eBus version folder named `eBus x.y.z`.
4. Select the `Source Code` folder.
5. Download the eBus source code by clicking on the file `eBusJavaSrc_x.y.z.tgz`.
6. Move the downloaded `eBusJavaSrc_x.y.z.tgz` to the location where you want to build eBus.
7. `gunzip` and `tar` extract `eBusJavaSrc_x.y.z.tgz` to create the `eBus` directory.
8. `cd` to the `eBus` directory.
9. Compile eBus: `$ ant compile`
10. Jar eBus: `$ ant jar`
11. Optionally test compiled eBus: `$ ant test`
    `DBConnectionTest` will fail due to a missing relational database required for the test.
    Other tests open TCP server ports which may fail due to the hard-coded port being in use.

Glossary

Index

Definition

Condition

eBus supports optional `net.sf.eBus.client.ECondition` to be associated with notification subscriptions and reply advertisements. Conditions restrict message delivery to those messages which satisfy the condition. For subscribers, this check is done just prior to actual message delivery. This means that potentially more messages are posted to the subscriber than are actually delivered.

For repliers, this check is done prior to posting the request message. This is necessary because eBus must determine if any repliers accept a request in order to return the correct request state to the requestor.

Dispatcher

Dispatcher asynchronously delivers messages to eBus clients. A Dispatcher consists of a client run queue and one or more threads. An application may configure multiple Dispatchers, each handling different client classes.

EClient

eBus creates one `EClient` instance for each application client, regardless of the number of interfaces the client implements or open feeds. The client maintains a *weak* reference to the application client, the undelivered message queue, and the client's open feeds. `EClient` connects the application client with the Dispatcher.

Feed Scope

An application defines a feed's scope when opening the feed. This scope defines the feed's visibility. That feed may be visible only within the local JVM, within both local and remote JVMs, and in remote JVMs only. For example, if a subscription feed has local-only scope, then it will receive notifications from local publisher feeds only. Notifications from remote publishers will not be forwarded to the local-only subscriber. The following table shows the interface between feed scopes:

Feed Scope

	Local Only	Local & Remote	Remote Only
Local Only	Match	Match	No match
Local & Remote	Match	Match	Match
Remote Only	No match	Match	No match

(Notice that a remote feed may only support a local & remote feed and not other remote only feeds.)

Feed scope gives the application developer control over how "far" a will go. If a notification message is intended to stay within a JVM, both the publish and subscribe feeds may be set to a local only scope. If a notification is meant for remote access only, the the publish feed is set to remote only scope. These examples also apply to request/reply feeds.

Message Key

eBus message feeds are uniquely identified by a "type+topic" message key. The "type" refers to a concrete message class. The "topic" is the unique message subject. Using "type+topic" allows a message subject to be used for multiple message classes.

Example: there is a message class named `CatalogUpdate` and a message subject "FishingGear". Combining the two results in the message key `CatalogUpdate`:"FishingGear". The message class may be combined with another subject to form another key `CatalogUpdate`:"CampingGear". Likewise, the subject may be combined with another message to form the key `SalesSpecial`:"FishingGear".

If you are familiar with subject-based addressing for message routing, then you know that these systems require subjects to be formatted according to a particular scheme. **eBus has no subject-formatting requirements.** You are free to format your message subjects any way you like.

Notification

An eBus message derived from `net.sf.eBus.messages.ENotificationMessage`. A notification message is posted by a `net.sf.eBus.client.EPublisher` instance and forwarded to all `net.sf.eBus.client.ESubscriber` instances currently subscribed to that notification message key.

Reply

An eBus message derived from `net.sf.eBus.messages.EReplyMessage`. A reply message is posted by a `net.sf.eBus.client.EReplier` instance in response to a request message and is sent directly to the `net.sf.eBus.client.ERequestor` which sent the request message.

Request

An eBus message derived from `net.sf.eBus.messages.ERequestMessage`. A request message is posted by a `net.sf.eBus.client.ERequestor` instance and forwarded to all `net.sf.eBus.client.EReplier` instances currently advertised for that request message key.