eBus Programmer's Manual

Version: 7.4.0

Released: September 5, 2024

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.

eBus (as of v. 5.0.0) is a Maven project deployed to Maven Central. All further releases will be available through Maven Central and may be accessed using the Maven dependency:

```
<dependency>
  <groupId>net.sf.ebus</groupId>
  <artifactId>core</artifactId>
  <version>7.4.0</version>
</dependency>
```

Gradle (long)

```
implementation group: 'net.sf.ebus', name: 'core', version: '7.4.0'
```

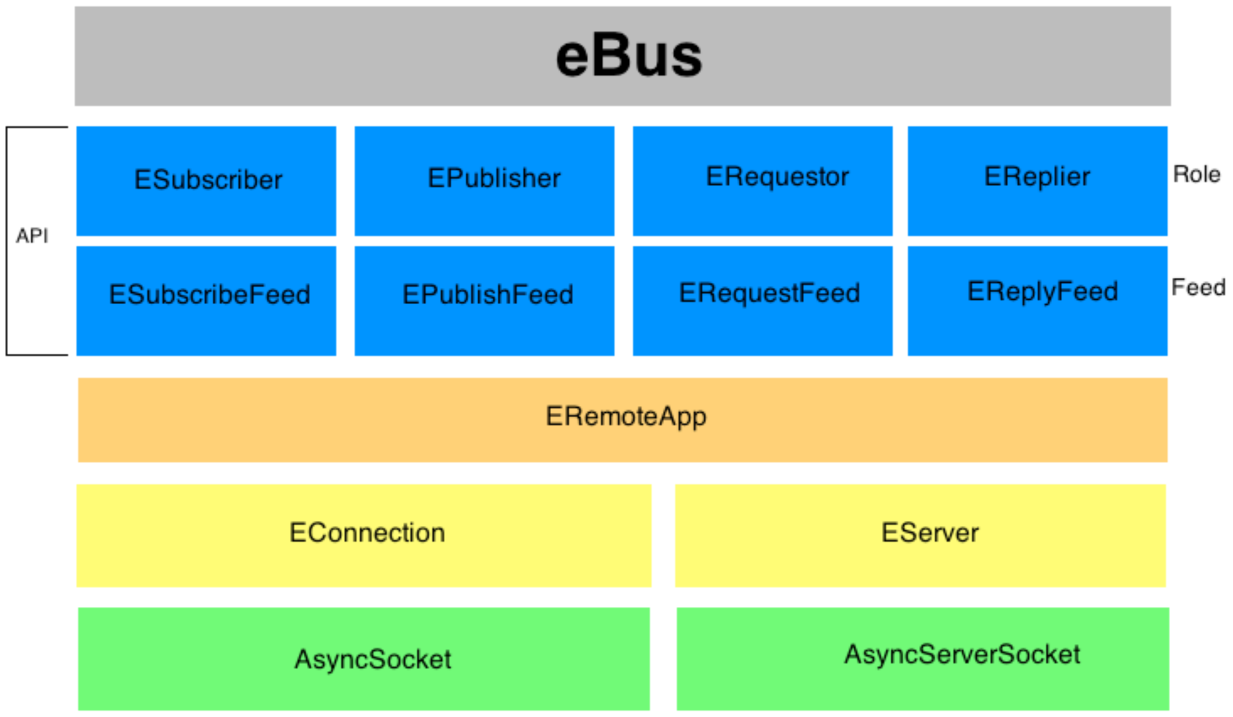
Gradle (short)

```
implementation 'net.sf.ebus:core:7.4.0'
```

Important! eBus v. 5.2.0 has significantly changed message class definition to better support message serialization/de-serialization. Please see the section ***Get the Message*** for detailed explanation regarding this change.

More Important! eBus v. 6.0.0 now requires Java 11. Previous versions are based on Java 1.8.

Overview



Application classes implement eBus roles and interact with eBus feeds. The publish/subscribe roles are `EPublisher` and `ESubscriber` and their respective feeds are `IEPublishFeed` and `IESubscribeFeed`. The request/reply roles are `ERequestor` and `EReplier` using `IERequestFeed` and `IEReplyFeed`.

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 "Feed me, Seymour!" takes the user beyond simple message feed types to complex feeds that simplify your application.

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. If you are using maven to build you project, then you can access eBus from a public repository (see above).

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.IEPublishFeed`, `IESubscribeFeed`, `IERequestFeed`, and `IEReplyFeed`.

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.

eBus release 6.1.0 deprecates the feed static `open(…)` method in favor a static `Builder` class. A `Builder` instance is accessed using the feed static `builder()` method. A feed instance is created via the `Builder.build()` method. This change does *not* apply to multi pattern feeds where the static `open()` method is still used.

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 shown in the final step.

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

public class CatalogPublisher
    implements EPublisher
{
    // EPublisher interface has one method.
    @Override public void publishStatus(final EFeedState feedState,
                                        final IEPublisherFeed 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.builder()).target(publisher)
                                                     .messageKey(key)
                                                     .scope(scope)
                                                     .build();
```

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 your `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
{
    // 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;

    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.builder()).target(this)
                                            .messageKey(mKey)
                                            .scope(mScope)
                                            .build();
            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.
    }
}
```

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 IEPublishFeed 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()) {
            mFeed.publish(
                (CatalogUpdate.builder()).subject(mKey.subject())
                                         .timestamp(Instant.now())
                                         .productName(productName)
                                         .price(price)
                                         .stockQty(stockQty)
                                         .build());
        }
    }
}
```

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 IEPublishFeed 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)) {
            // 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

```java
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.client.IEPublishFeed;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ENotificationMessage;

public class CatalogPublisher
    implements EPublisher
{
    // 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;

    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.builder()).target(this)
                                            .messageKey(mKey)
                                            .scope(mScope)
                                            .build();
            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 IEPublishFeed 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()) {
            mFeed.publish(
               (CatalogUpdate.builder()).subject(mKey.subject())
                                        .timestamp(Instant.now())
                                        .productName(productName)
                                        .price(price)
                                        .stockQty(stockQty)
                                        .build());
        }
    }

    // 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.
    }
}
```

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.IESubscribeFeed;
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 IESubscribeFeed feed) {
         See step 3.
    }

    @Override public void notify(final ENotificationMessage msg,
                                 final IESubscribeFeed 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.I.builder()).target(subscriber)
                                                       .messageKey(key)
                                                       .scope(scope)
                                                       .condition(condition)
                                                       .build();
```

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.client.IESubscribeFeed;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ENotificationMessage;

public class CatalogSubscriber
    implements ESubscriber {
    // 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;

    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; defaults to null.
            mFeed = (ESubscribeFeed.builder()).target(this)
                                              .messageKey(mKey)
                                              .scope(mScope)
                                              .build();
            mFeed.subscribe();
        } catch(IllegalArgumentException argex) {
            // Feed open failed. Place recovery code here.
        }
    }

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

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

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

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 are 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 IESubscribeFeed 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 IESubscribeFeed feed) {
        Notification handling code here.
    }
}
```

Step 5: Unsubscribing

A subscriber has two ways to retract a subscription:

1. `IESubscribeFeed.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. `IESubscribeFeed.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

```java
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.client.IESubscribeFeed;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ENotificationMessage;

public class CatalogSubscriber
    implements ESubscriber {
    // 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;

    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; defaults to null.
            mFeed = (ESubscribeFeed.builder()).target(this)
                                              .messageKey(mKey)
                                              .scope(mScope)
                                              .build();
            mFeed.subscribe();
        } catch(IllegalArgumentException argex) {
            // Feed open failed. Place recovery code here.
        }
    }

    @Override public void feedStatus(final EFeedState feedState,
                                     final IESubscribeFeed 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 IESubscribeFeed feed) {
        Notification handling code here.
    }

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

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.IEReplyFeed;
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 IEReplyFeed feed) {
        See step 3.
    }

    @Override public void cancelRequest(final EReplyFeed.ERequest request,
                                        final boolean 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, advertising the replier, and finally marking the feed state as up:

```
replyFeed = (net.sf.eBus.client.EReplyFeed.builder()).target(replier)
                                                     .messageKey(key)
                                                     .scope(scope)
                                                     .condition(condition)
                                                     .build();
```

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
{
    // 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;

    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; defaults to null.
            mFeed = (EReplyFeed.builder()).target(this)
                                          .messageKey(mKey)
                                          .scope(mScope)
                                          .build();
            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.
    }
}
```

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 IEReplyFeed 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 boolean mayRespond) {
         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:

```
    EReplier.cancelRequest(ERequestFeed.ERequest request, boolean mayRespond);
```

When `mayRespond` is `false` this means the request is unilaterally canceled and is no longer active. The replier may no longer reply to the request and should stop any processing with respect to the canceled request.

When `mayRespond` is `true` this means that the replier *may* respond to the request and either accept the cancellation or reject it. In both cases the replier should send a `net.sf.eBus.messages.EReplyMessage` back to the requester with replyStatus set to either `EReplyMessage.ReplyStatus.CANCELED` (cancel request accepted) or `EReplyMessage.ReplyStatus.CANCEL_REJECT` (cancel request rejected). If the cancellation is accepted then the replier should clean up the request as the request is now no longer active *from the replier's perspective*.

If the cancellation is rejected then the request is still active and further replies may be posted to the request.

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

public class CatalogReplier
    implements EReplier
{
    @Override public void cancelRequest(final EReplyFeed.ERequest request,
                                        final boolean mayRespond) {
        try {
            // Throws an exception if the request cannot be stopped.
            stopOrderProcessing(request);
            mRequests.remove(request;

            // Reply to an optional cancellation.
            if (mayRespond) {
                cancelReply(ReplyStatus.CANCELED, null);
            }
        } catch (Exception jex) {
            // If request processing could not be stopped and this replier may
            // respond, then send back a cancel reject.
            if (mayRespond) {
                cancelReply(ReplyStatus.CANCEL_REJECT, jex.getMessage());
            }
        }
    }

    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) && request.isActive()) {
            request.reply((OrderReply.builder()).subject(mKey.subject)
                                                   .timestamp(Instant.now())
                                                   .replyStatus(status)
                                                   .replyReason(reason)
                                                   .build());

            // If the request processing is complete, remove the request.
            if (status.isFinal()) {
                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 `unadvertise`, 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();

        // While eBus will does this for us, it is better to do it ourselves.
        for (EReplyFeed.ERequest request : mRequests) {
            request.reply((EReplyMessage.builder()).subject(subject)
                                                   .timestamp(Instant.now())
                                                   .replyStatus(ReplyStatus.ERROR)
                                                   .replyReason("shutting down")
                                                   .build());
        }

        mRequests.clear();

        if (mFeed != null) {
            // close() implicitly unadvertise 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.client.IEReplyFeed;
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
{
    // 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.
    private final List<EReplyFeed.ERequest> mRequests;

    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; defaults to null.
            mFeed = (EReplyFeed.builder()).target(this)
                                          .messageKey(mKey)
                                          .scope(mScope)
                                          .build();
            mFeed.advertise();
            mFeed.updateFeedState(EFeedState.UP);
        } catch (IllegalArgumentException argex) {
            // Advertisement failed. Place recovery code here.
        }
    }
```

```
@Override public void request(final EReplyFeed.ERequest request,
                              final IEReplyFeed 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 boolean mayRespond) {
    try {
        // Throws an exception if the request cannot be stopped.
        stopOrderProcessing(request);
        mRequests.remove(request;

        // Reply to an optional cancellation.
        if (mayRespond) {
            cancelReply(ReplyStatus.CANCELED, null);
        }
    } catch (Exception jex) {
        // If request processing could not be stopped and this replier may
        // respond, then send back a cancel reject.
        if (mayRespond) {
            cancelReply(ReplyStatus.CANCEL_REJECT, jex.getMessage());
        }
    }
}

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

    if (mRequests.contains(request) && request.isActive()) {
        request.reply((OrderReply.builder()).subject(mKey.subject)
                                            .timestamp(Instant.now())
                                            .replyStatus(status)
                                            .replyReason(reason)
                                            .build();

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

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

        // While eBus will does this for us, it is better to do it ourselves.
        for (EReplyFeed.ERequest request : mRequests) {
            request.reply((EReplyMessage.builder()).subject(subject)
                                                   .timestamp(Instant.now())
                                                   .replyStatus(ReplyStatus.ERROR)
                                                   .replyReason("shutting down")
                                                   .build());
        }

        mRequests.clear();

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

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.client.IERequestFeed;
import net.sf.eBus.messages.EReplyMessage;

public class CatalogRequestor
    implements ERequestor
{
    // ERequestor interface has two methods.
    @Override public void feedStatus(final EFeedState feedState,
                                     final IERequestFeed 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.builder()).target(requestor)
                                               .messageKey(key)
                                               .scope(scope)
                                               .build();
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
{
    private final EMessageKey mKey;
    private final FeedScope mScope;
    private final List<ERequestFeed.ERequest> mRequests;
    private ERequestFeed mFeed;

    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.builder().target(this)
                                           .messageKey(mKey)
                                           .scope(mScope)
                                           .build();
            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.
    }
}
```

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 IERequestFeed 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 = (CatalogOrder.builder()).subject(mKey.subject())
                                                        .timestamp(Instant.now())
                                                        .product(product)
                                                        .quantity(quantity)
                                                        .price(price)
                                                        .shipping(shipping)
                                                        .address(address)
                                                        .build();

        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.

eBus release 5.6.0 introduced a two new features with respect to canceling an active request:

1. `net.sf.eBus.messages.EReplyInfo` annotation has a new element: `boolean mayClose`. This element defaults to `true` which means that `ERequestFeed.ERequest.close()` may be used to cancel an active request. If set to `false` then `ERequest.close()` may *not* be called (results in an `IllegalStateException` being thrown). This leads to the second change.

2. A new method `ERequestFeed.ERequest.cancel()` is added which is an *optional* cancel request. Optional means that active repliers to this request may respond to this cancel request. Those cancel replies will most likely be `EReplyMessage` instances with a `replyStatus` set to either `ReplyStatus.CANCELED` (cancel request accepted) or `ReplyStatus.CANCEL_REJECT` (cancel request rejected). If all repliers accept the cancel request then the request is no longer active. If any one replier rejects the cancel request then the request is still active and the requester may expect further replies to the request.

```
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.client.IERequestFeed;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.EReplyMessage;

public class CatalogRequestor
    implements ERequestor
{
    private final EMessageKey mKey;
    private final FeedScope mScope;
    private final List<ERequestFeed.ERequest> mRequests;
    private ERequestFeed mFeed;

    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.builder()
                   .target(this).messageKey(mKey).scope(mScope).build();
            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 IERequestFeed 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 = (CatalogOrder.builder()).subject(mKey.subject())
                                                         .timestamp(Instant.now())
                                                         .product(product)
                                                         .quantity(quantity)
                                                         .price(price)
                                                         .shipping(shipping)
                                                         .address(address)
                                                         .build();
        try {
            synchronized (mRequests) { mRequests.add(mFeed.request(msg)); }
        } catch (Exception jex) {
            // Request failed. Put recovery code here.
        }
    }
}
```

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, IESubscribeFeed).
```

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 `IESubscribeFeed.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.

Note: as of eBus release 7.1.0, the notify callback method's first parameter may be declared as the target `ENotificationMessage` subclass. In the following example code, the callback method `latestUpdate` declares the message type as `CatalogUpdate`. This means there is no longer the need to declare the type as `ENotificationMessage` and downcast to the target type.

```
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.client.IESubscribeFeed;
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.builder())
                    .target(this)
                    .messageKey(mKey1),
                    .scope(mScope1)
                    .condition(m -> ((AppMessage) m).value >= 100))
                    .notifyCallback(
                        (msg, feed) -> { /* Put notify callback code here */ })
                    .build();
            mFeed1.subscribe();

            mFeed2 = (ESubscribeFeed.builder()).target(this)
                                                .messageKey(mKey2)
                                                .scope(mScope2)
                                                .notifyCallback(this::latestUpdate)
                                                .build();
            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 IESubscribeFeed feed) {
        // Status updates for all feeds handled in the same way.
    }

    private void latestUpdate(final CatalogUpdate msg,
                              final IESubscribeFeed 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:

```
@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);

// Open the request feed, setting status and reply callbacks.
// Note: message key must be set prior to setting callbacks for specific reply message
// classes.
orderFeed = (ERequestFeed.builder()).target(this)
                                    .messageKey(requestKey)
                                    .scope(EFeed.FeedScope.REMOTE)
                                    .statusCallback(this::requestStatus)
                                    .replyCallback(this::orderReply)
                                    .replyCallback(OptionOrderState.class,
                                                   this::orderState)
                                    .replyCallback(OptionOrderFill.class,
                                                   this::orderFill)
                                    .build();
```

Hybrid Object Pattern

When using eBus in a major application the central eBus objects will contain so many feeds that the object becomes convoluted and difficult to understand. As an example consider an eBus object implementing a trading algorithm. This object implements the following roles:

- `EReplier`: receives algo requests to buy or sell an instrument using the algorithm's logic.
- `ERequestor`: algorithm places order on one or more exchanges in order to satisfy its own request.
- `ESubscriber`: algorithm subscribes to market data, dynamic configuration changes to algorithm parameters, instrument and exchange dynamic performance statistics used to guide placing exchange orders.
- `EPublisher`: algorithm publishes its own dynamic performance statistics.

Placing all these feeds and associated data members into a single eBus object results in code confusion, making it difficult to separate the essential algorithm data and logic from subsidiary information supporting the algorithm. This section describes a hybrid object pattern useful for reducing this code confusion.

Firstly, what is a hybrid object? There are two types of classes: active and passive. A passive object does not initiate actions but is only acted upon. Example passive classes are `java.util.ArrayList` or `java.time.Duration`. Instances of these classes do nothing until other code initiates a method call to the instance. A passive object is contained within an active object.

An active object can initiate action by sending an eBus message. An active object does not need to wait to be acted upon before acting. An active object does not exist within another object, standing alone within an application.

A hybrid object lies between active and passive. A hybrid object sends and/or receives messages but is contained within an active object. This active object treats the hybrid object as a passive object, interacting with the hybrid object using method calls. Please note that hybrid objects should *not* be shared among active objects.

Back to the trading algorithm active object. The following example concentrates on the market data hybrid object only but can be readily expanded to handle the feeds as well.

```
public final class MakeMoneyAlgo
    implements EReplier, ERequestor, ESubscriber, EPublisher {

    // Hybrid objects created on active object start up.
    private MarketData mMktData;
    private MakeMoneyConfig mAlgoConfig;
    ...

    private EReplyFeed mOrderFeed;

    public MakeMoneyAlgo(...) { ... }

    @Override public void startup() {
        // Create hybrid instances and then have hybrid create its feeds.
        // Note that this MakeMoneyAlgo reference is passed to hybrid object
        // constructor.
        mMktData = new MarketData(this, ...);
        mMktData.startup();

        mOrderFeed = (EReplyFeed.builder()).target(this)
                                           .messageKey(order request key)
                                           .scope(EFeed.FeedScope.REMOTE)
                                           .requestCallback(this::onOrder)
                                           .build();
        mOrderFeed.advertise();
        mOrderFeed.updateFeedState(EFeedState.UP);
    }

    private void onOrder(final EReplyFeed.ERequest request,
                         final IEReplyFeed feed) {
        final OrderRequest newOrder = (OrderRequest) request;
        final OrderBook book =
            mMktData.getOrderBook(newOrder.instrument, mAlgoConfig.getBookDepth());

        // Validate and process order request using information retrieved from hybrid
        // objects.
    }
}
```

Note that `MakeMoneyAlgo` implements the `EReplier`, `ERequestor`, `ESubscriber`, and `EPublisher` interfaces and *not* the hybrid objects. This is because *only active objects may implement eBus role interfaces*. This includes the `EObject` interface.

Passing the `MakeMoneyAlgo` reference to the `MarketData` constructor is key to any hybrid object:

```
public final class MarketData {

    // MakeMoneyAlgo instance containing this hybrid object.
    private final MakeMoneyAlgo mOwner;

    // Market data subscription made on start up.
    private ESubscribeFeed mMktDataFeed;

    // Place order book data members here.

    public MarketData(final MakeMoneyAlgo owner, ...) {
        mOwner = owner;
        ...
    }

    public void startup() {
        // Note: hybrid feed uses MakeMoneyAlgo reference as target but uses
        // MarketData methods for feed status and message delivery.
        // This is what makes MarketData a hybrid object.
        mMktDataFeed = (ESubscribeFeed.builder()).target(mOwner)
                                                 .messageKey(market data message key)
                                                 .scope(EFeed.FeedScope.REMOTE)
                                                 .statusCallback(this::onFeedStatus)
                                                 .notifyCallback(this::onMarketData)
                                                 .build();
        mMktDataFeed.subscribe();
    }

    public OrderBook getOrderBook(final Instrument instrument, final int depth) {
        // Returns instrument's order book up to the given depth. MakeMoneyAlgo uses
        // this order book combined with other parameters to decide what orders to
        // place on what exchanges.
    }

    private void onFeedStatus(final EFeedState state, final IESubscribeFeed feed) {
        // Place feed status update code here especially when market data feed goes
        // down.
    }

    private void onMarketData(final ENotificationMessage msg,
                              final IESubscribeFeed feed) {
        // Update market data members with latest update.
    }
}
```

See Dispatcher section for why hybrid object feeds must target its active object owner.

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.

Note: As of eBus v. 5.2.0 message definition has fundamentally changed. Please read this section. eBus now uses the builder pattern when de-serializing. eBus orders fields by name and size starting with fields of fixed size from 8 bytes to 1 bytes followed by variable-sized fields.

[2] POJO: Plain Old Java Object.

Defining an eBus Message

Step 0: eBus supported message field types

eBus message fields must either an eBus-defined type or a user-defined type (shown in step 6). The eBus-supported 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. `java.util.UUID`,
15. `org.decimal4j.api.Decimal` interface subtypes,
16. `net.sf.eBus.messages.EField`[3] user-defined subclass,
17. `net.sf.eBus.messages.EFieldList`[4],
18. `net.sf.eBus.messages.EMessageList`, and
19. `net.sf.eBusx.geo EFIeld` implementations of GeoJSON objects.

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

See Appendix A: Binary Message Layout to learn how the above field types are serialized.

[3] See section "Step 6: Message field type definition" for more information on user-defined fields.

[4] 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[5] `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 6.
    public final int quantity;
    private static final long serialVersionUID = 1L;
}
```

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

Step 2: Message Annotation

`@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}, mayClose=false)`, the class `EReplyMessage` may be sent in reply to any request.

The attribute `mayClose=false` means that this request may *not* canceled using the method `ERequestFeed.ERequest.close()` - which unilaterally cancels the request. Instead the method `ERequest.cancel()` may be used which asks active repliers to accept the request cancellation. These repliers may accept or reject this cancellation. If all repliers accept then the request is canceled. Otherwise the request is still active. Attribute `mayClose` may be overridden by `ERequestMessage` subclasses. So while `ERequestMessage` sets `mayClass` to `false`, a subclass can set it this attribute to `true`.

See Step 4: Canceling a request for more information about optional request cancellation from the replier perspective.

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.EReplyInfo;
import net.sf.eBus.messages.ERequestMessage;

@EReplyInfo(replyMessageClasses={CatalogOrderReply.class}, mayClose=false)
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: Static Message builder() method

Every eBus message subclass must provide a builder used to construct the target message instance. This builder is provided by the following method:

```
public static builder-class builder()
```

where `builder-class` is defined in step 4.

Note: Class `CatalogOrder` has five fields total: `subject`, `timestamp`, `orderId`, `price`, and `quantity`.

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

@EReplyInfo (replyMessageClasses={CatalogOrderReply.class}, mayClose=false)
public final class CatalogOrder
    extends ERequestMessage
    implements Serializable {

    public final String orderId;
    public final Money price;  See step 6.
    public final int quantity;
    private static final long serialVersionUID = 1L;

    // Step 5 shows how a message class is constructed from the builder.

    // This method must be named "builder" and be public static.
    public static OrderBuilder builder() {
        return (new OrderBuilder());
    }

// Inner class OrderBuilder defined next.
}
```

Step 4: Builder Inner Class

The Builder subclass returned by the `public static builder()` method is responsible for recreating the target message class instance from the de-serialized fields. This class must `extend` the Builder class corresponding to the target message class. When building an ENotificationMessage sub-class, then the builder class must `extend ENotificationMessage.Builder<`*`target-class, builder-class`*`>`. The same applies to ERequestMessage and EReplyMessage. Note that the Builder base classes take two generic parameters: target message class (`M`) and builder leaf class (`B`). The first parameter is used to define the `M build()` method which generates the target message instance from the configured message parameters. The second parameter is used to return the correct builder class when a message field is defined in a super class. This allows chaining together parameter set method even when the setters are defined at different levels of the builder class hierarchy.

The builder instance is the sole argument to the message class constructor (see step 5).

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

@EReplyInfo (replyMessageClasses={CatalogOrderReply.class}, mayClose=false)
public final class CatalogOrder
    extends ERequestMessage
    implements Serializable {

    // Builder class should be public so it can be accessed by all other classes
    // and static because it does not share its owning class' members.
    public static final class OrderBuilder
        extends ERequestMessage.Builder<CatalogOrder, OrderBuilder> {

            // Builder has the same data member as the target class - but not public
            // and final.
            private String mOrderId;
            private Money mPrice;
            private int mQuantity;

            // Constructor shown in Step 4a.

            // Setter methods shown in Step 4b.

            // Abstract method overrides shown in Step 4c.
    }
}
```

Step 4a: Builder Inner Class Constructor

The builder class constructor takes zero arguments and is `private` because only the `public static builder()` method in the encapsulating class can create a builder class instance. The builder class constructor passes the *`target-class`*`.class` instance to the builder super class constructor. This class instance is used to create a `net.sf.eBus.messages.ValidationException` when the builder detects that an invalid target message was configured causing the `build()` method to fail.

```
import net.sf.eBus.messages.ERequestMessage;

// Builder class should be public so it can be accessed by all other classes
// and static because it does not share its owning class' members.
public static final class OrderBuilder
    extends ERequestMessage.Builder<CatalogOrder, OrderBuilder> {

        private String mOrderId;
        private Money mPrice;
        private int mQuantity;

        // private constructor so only public static OrderBuilder builder()
        // method can instantiate this class.
        private OrderBuilder() {
            super (CatalogOrder.class);
        }

        // Setter methods shown in Step 4b.

        // Abstract method overrides shown in Step 4c.
}
```

Step 4b: Builder Inner Class Setter Methods

There must be one setter for each `public final` message field *in the encapsulating class*[6] with the signature:

```
public builder-class field-name(final field-type value)
```

where:

`builder-class` is the builder class name,
`field-name` *exactly* matches the message field name, and
`field-type` matches the message field type.

Failure to correctly define a setter message for all the encapsulating class' message fields results in a `net.sf.eBus.messages.InvalidMessageException` thrown by `DataType.findType(Class)` method.

The setter method is responsible for checking if the provided value is valid by itself. Override the `validate` method to check if the overall configuration is acceptable.

```
import net.sf.eBus.messages.ERequestMessage;

public static final class OrderBuilder
    extends ERequestMessage.Builder<CatalogOrder, OrderBuilder> {
    // Constructor shown in Step 4a.

    public OrderBuilder orderId(String id) {
        if (id == null || id.isEmpty()) {
            throw (new IllegalArgumentException("id is null or empty"));
        }
        mOrderId = id;
        return (this);
    }
    public OrderBuilder price(Money px) {
        if (money == null) {
            throw (new NullPointerException("px is null"));
        }
        mPrice = px;
        return (this);
    }
    public OrderBuilder quantity(int qty) {
        if (qty <= 0) {
            throw (new IllegalArgumentException("qty < zero"));
        }
        mQuantity = qty;
        return (this);
    }

    // Abstract method overrides shown in Step 4c.
}
```

[6] Super class message fields are set by the super class' builder.

Step 4c: Builder Inner Class Method Overrides

The builder class is required to override the abstract method `protected` *`target-class`* `buildImpl()`. This method returns a new *`target-class`* instance based on the builder's field settings.

The `protected Validator validate(Validator problems)` method may the overridden so pre-build validation may be done. This is necessary to check if required fields are set or that the overall message configuration is correct. One example is setting one field to a particular value may restrict another fields setting. This cannot and should not be done in the individual field setters but in the overall message validation. Multi-field validation is performed using `Validator.requireTrue(BiPredicate<V1, V2>, V1 value1, V2 value2, String fieldName1, String fieldName2, String message)` method.

Note: when using `Validator.requireTrue()` methods, be sure that the `Predicate` argument returns `true` only if the tested fields *are valid*. If the predicate returns `false`, `Validator` marks the field(s) as invalid with the given reason.

When an invalid setting is detected add a string value to the problems list succinctly explaining the problem and, optionally, how to correct the problem.

```
import net.sf.eBus.messages.ERequestMessage;

public static final class OrderBuilder
    extends ERequestMessage.Builder<CatalogOrder, OrderBuilder> {
    // Constructor shown in Step 4a.
    // Setter methods shown in Step 4b.
    @Override protected CatalogOrder buildImpl() {
        return (new CatalogOrder(this));
    }
    @Override protected Validator validate(Validator problems) {
        // Must call base class validate first.
        return (super.validate(problems)
                     .requireNotNull(mOrderId, "orderId")
                     .requireNotNull(mPrice, "price")
                     .requireTrue(v -> (v > 0),
                                  mQuantity,
                                  "quantity",
                                  Validator.NOT_SET)
                     // Total order cost cannot exceed a maximum allowed value.
                     .requireTrue(
                          (v1, v2) ->
                              ((v1.multiply(v2)).compareTo(
                                 MAX_ALLOWED_VALUE) <= 0),
                          mPrice,
                          mQuantity,
                          "price",
                          "quantity",
                          "order exceeds maximum allowed value"));
    }
}
```

Step 5: Message Object Constructor

Message class and builder are tied together by a single private constructor `message-class(message-builder builder)` where the constructor initializes its fields from the builder settings. This constructor is private because only the builder inner classes calls this constructor.

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

@EReplyInfo (replyMessageClasses={CatalogOrderReply.class}, mayClose=false)
public final class CatalogOrder
    extends ERequestMessage
    implements Serializable {

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

    // Private constructor because only OrderBuilder may call this constructor.
    private CatalogOrder(OrderBuilder builder) {
        super (builder);

        this.orderId = builder.mOrderId;
        this.price = builder.mPrice;
        this.quantity = builder.mQuantity;
    }
}
```

Step 6: 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.

There is a performance loss using abstract message fields. eBus includes the leaf field class name in the serialization. The de-serialization code reads in the name and passes it to `Class.forName(String)` Using abstract `EField` subclass slows the message transmission. This is not done when using a concrete `EField` subclass as a message field since the field type is fixed.

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

public final class Money
    extends EField
    implements Serializable {

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

    public Money(final MoneyBuilder builder) {
        super (builder);

        this.price = builder.mPrice;
        this.currency = builder.mCurrency;
        this.qty = builder.mQty;
    }

    public static MoneyBuilder builder() {
        return (new MoneyBuilder());
    }

    public static final class MoneyBuilder
        extends EField.Builder<Money, MoneyBuilder>
    {
        // This class is defined in the same manner as OrderBuilder.
    }
}
```

Step 7: Complete Message Class

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

@EReplyInfo (replyMessageClasses={CatalogOrderReply.class}, mayClose=false)
public final class CatalogOrder
    extends ERequestMessage
    implements Serializable
{
    // 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;
    public final int quantity;
    private static final long serialVersionUID = 1L;

    // Private constructor because only OrderBuilder may call this constructor.
    private CatalogOrder(OrderBuilder builder) {
        super (builder);

        this.orderId = builder.mOrderId;
        this.price = builder.mPrice;
        this.quantity = builder.mQuantity;
    }
    // This method must be named "builder" and be public static.
    public static OrderBuilder builder() {
        return (new OrderBuilder());
    }

    // Builder class should be public so it can be accessed by all other classes
    // and static because it does not share its owning class' members.
    public static final class OrderBuilder
        extends ERequestMessage.Builder<CatalogOrder, OrderBuilder> {
        // Builder has the same data member as the target class - but not public and
        // final.
        private String mOrderId;
        private Money mPrice;
        private int mQuantity;

        // private constructor so only public static OrderBuilder builder() method can
        // instantiate this class.
        private OrderBuilder() {
            super (CatalogOrder.class);
        }

        public OrderBuilder orderId(String id) {
            if (id == null || id.isEmpty()) {
                throw (new IllegalArgumentException("id is null or empty"));
            }

            mOrderId = id;
            return (this);
        }
        public OrderBuilder price(Money px) {
            if (money == null) {
                throw (new NullPointerException("px is null"));
            }

            mPrice = px;
            return (this);
        }
        public OrderBuilder quantity(int qty) {
            if (qty <= 0) {
                throw (new IllegalArgumentException("qty < zero"));
            }

            mQuantity = qty;
            return (this);
        }
        @Override protected CatalogOrder buildImpl() {
            return (new CatalogOrder(this));
        }
        @Override protected Validator validate(Validator problems) {
            // Must call base class validate first.
            return (super.validate(problems)
                          .requireNotNull(mOrderId, "orderId")
                          .requireNotNull(mPrice, "price")
                          .requireTrue(v -> (v > 0),
                                       mQuantity,
                                       "quantity",
                                       Validator.NOT_SET)
                          // Total order cost cannot exceed a maximum allowed value.
                          .requireTrue(
                               (v1, v2) ->
                                   ((v1.multiply(v2)).compareTo(
                                      MAX_ALLOWED_VALUE) <= 0),
                               mPrice,
                               mQuantity,
                               "price",
                               "quantity",
                               "order exceeds maximum allowed value"));
        }
    }
}
```

Defining an Extendable Field

The above example assumes the user-defined message is final. But what if message or field may be both instantiated or extended? Then that message or field must provide both an abstract and a concrete builder. Consider the following user-defined fields:

```
public class UserInfo
    extends EField
{
    public final String name;
    public final int age;

    private UserInfo(final Builder<?, ?> builder) {
        super (builder);

        this.name = builder.mName;
        this.age = builder.mAge;
    }

    // Builder code shown below.
}

public final class Employee
    extends UserInfo
{
    public final String department;

    private Employee(final Builder builder) {
        super (builder);

        this.department = builder.mDepartment;
    }

    // Builder code show below.
}
```

Class `UserInfo` can be instantiated on its own and serves as `Employee` superclass. This means `UserInfo` needs to provide one builder for constructing `UserInfo` instances and another abstract builder to server as `Employee.Builder` superclass. The `UserInfo` abstract builder is now defined.

```
public static abstract class Builder<M extends UserInfo,
                                     B extends Builder<M, ? extends UserInfo.Builder>>
    extends EField.Builder<M, B>
{
    private String mName;
    private int mAge;

    protected Builder(final Class<? extends EMessageObject> targetClass) {
        super (targetClass);

        this.age = -1;
    }

    public B name(final String value) {
        if (value == null || value.isEmpty()) {
            throw (new IllegalArgumentException("value is null or empty"));
        }

        this.mName = value;
        return ((B) this);
    }

    public B age(final int age) {
        if (age < 0) {
            throw (new IllegalArgumentException("age < zero"));
        }
        this.mAge = age;
        return ((B) this);
    }

    @Override protected Validator validate(final Validator problems) {
        return (super.validate(problems)
                     .requireNotNull(mName, "name")
                     .requireTrue(v -> (v >= 0), mAge, "age", Validator.NOT_SET));
    }
}
```

Note that unlike a concrete builder an abstract builder must define the `M` and `B` class parameters. These parameters match those in `EField.Builder` but are restricted to `UserInfo` and `UserInfo.Builder`. The `B` builder parameter is used to downcast setter method return values to the concrete builder type. Further the abstract builder does not override `buildImpl` method because only concrete builders create the target class instance.

Given this abstract builder the concrete builder is defined as:

```
public static final class ConcreteBuilder
    extends Builder<UserInfo, ConcreteBuilder>
{
    private ConcreteBuilder() {
        super (UserInfo.class);
    }

    @Override protected UserInfo buildImpl() {
        return (new UserInfo(this));
    }
}
```

Since the abstract `Builder` class defines the setters and validation, the only left for `ConcreteBuilder` is to define `buildImpl`. The final code to define is the `UserInfo` `builder` method.

```
public static UserInfo.Builder<?, ?> builder() {
    return (new ConcreteBuilder());
}
```

A `ConcreteBuilder` instance is returned because the caller wants to build a `UserInfo` instance.

The `Employee` builder extends the abstract `UserInfo.Builder` class and defines the `department` setter method and `validate`, `buildImpl` overrides.

```
public static final class Builder extends UserInfo.Builder<Employee, Builder> {
    private String mDepartment;

    private Builder() { super (Employee.class); }

    public Builder department(final String value) {
        if (value == null || value.isEmpty()) {
            throw (new IllegalArgumentExpression(""));
        }

        this.mDepartment = value;
        return (this);
    }

    @Override protected void validate(final List<String> problems) {
        super.validate(problems);

        if (mDepartment == null) { problems.add("department not set");}
    }

    @Override protected Employee buildImpl() { return (new Employee(this)); }
}
```

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.[7]

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`.

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

Local eBus Messages

By default eBus requires all message fields to be serializable.[8] Since eBus does *not* support `java.util.Map` it cannot be used as a message field since eBus does not know how to serialize a map for transmission to another eBus application.

But what if a message is meant for use within a JVM only? That changes things since that message does not need to be serialized. In that case, add the `@ELocalOnly` annotation to the message class definition. Message fields may then be any type desired. Further the message class does not need to define a `public static builder-class builder()` method or the `builder-class` inner class. But message fields *must* still be `public final`.

Note: while the below constructor currently works, `ENotificationMessage(String subject, long timestamp)` constructor is deprecated. In the future, local-only messages will be required to use a builder because `ENotificationMessage` will only have a builder-based constructor.

```
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import net.sf.eBus.messages.ELocalOnly;
import net.sf.eBus.messages.ENotificationMessage;

@ELocalOnly
public final class LocalMessage
    extends ENotificationMessage
    implements Serializable
{
    // Map is not an eBus-supported type but allowed in this local message.
    public final Map<String, Integer> studentIds;

    private static final long serialVersionUID = 0x1L;

    public LocalMessage(final String subject,
                        final long timestamp,
                        final Map<String, Integer> ids) {
        super (subject, timestamp);

        studentIds = Collections.unmodifiableMap(new HashMap<>(ids));
    }
}
```

[8] See Appendix A for eBus supported field types.

Message Field Annotations

The following annotations may be applied to the specified field data types.

String: @EStringInfo(charset = "*charset name*", *lineCount = n, maximumAllowedSize = n)*

eBus uses the `java.nio.charset.StandardCharsets.UTF_8` as the default character set to serialize, de-serialize Java `String` fields. This default character set can be overridden using the `@EStringInfo` annotation for `String` message fields:

```
@EStringInfo(charset = "latin1") public final String lastName;
```

where `charset` should be a character set name or alias known to `java.nio.charset.Charset.forName` method. If `forName` throws an exception for the given `charset` name, then eBus will *quietly* use `UTF_8` instead. No exception will be thrown or error logged.

`@EStringInfo` has a second attribute `lineCount` used to specify the number of lines that may be in the string field. If not defined, the line count defaults to a single line. lineCount is *not* used by eBus and is provided solely for application use. One possible use for this attribute is deciding whether to use a JavaFX `TextField` to input a single line string field or a `TextArea` to input a multi-line string field.

The third attribute is used to specify the maximum length allowed for the text field. Field serialization fails with a `BufferOverflowException` if the `String`'s length exceeds this value.

Array, EFieldList, EMessageList, BigInteger: @EArrayInfo(*maximumAllowedSize = n)*

Specifies the allowed maximum size of the serialized field. If the array or list size exceeds this value, then field serialization fails with a `BufferOverflowException`.

All: @FieldDisplayIndex(index = n)

eBus 5.3.2 introduced the annotation `@FieldDisplayIndex` which is used to define field ordering for *display* purposes. The problem is that `MessageType.fields()` returns fields in *serialization* order. This ordering is meant for superior serialization performance but this ordering does not make human sense. This is corrected by `MessageType.displayFields()`. The returned field ordering is from base class `EMessageObject` fields down to the leaf class. Meaning super class fields are listed first, the leaf message class last.

Usage: `FieldDisplayIndex(index = n)` where `n` >= zero. Field display indices do not have to start at zero or have to be strictly sequential but must be in increasing order. If multiple field display indices have the same value, the ordering is undefined among those fields.

Compiling Messages

When an eBus messages is sent across a connection to a remote eBus application, that must be serialized by the sending application and de-serialized by the receiving application. The code performing this serialization, de-serialization is generated by eBus as run-time based on the eBus message class. eBus provides two messages compilers: `JavaMessageCompiler` and `InvokeMessageCompiler`.

`JavaMessageCompiler` uses the `javassist` package to compile and load a `DataType`-subclass generated specifically to serialize, de-serialize an eBus message fields. This technique provides the fastest and memory-efficient message serialization, de-serialization. That is why it is the default message compiler.

But if `javassist` is not available, then use `InvokeMessageCompiler`. This creates a `InvokeMessageType` class instance to serialize, de-serialize a given eBus message class. As the name implies, `InvokeMessageType` uses the `java.lang.invoke` package to access message fields when serializing and message builder set methods when de-serializing. Changing the message compiler type ***must*** be done by setting the property `messageCompiler = JAVA_INVOKE` in an eBus configuration file and then passing that file in the command line option `-Dnet.sf.eBus.config.jsonFile=<file>`.

Key to eBus

Message keys link eBus roles to eBus messages. A publisher, subscriber, requester, replier clients open a feed, it specifies a message key. If that key is not already defined in the eBus message key dictionary when the feed is opened, then it is added to the dictionary. But what message keys are already defined in the eBus dictionary?

eBus release 4.5.0 makes it possible to both retrieve and add message keys apart from opening `EFeeds`. There are two methods provided for added message keys to the dictionary: `EFeed.addKey(EMessageKey)` and `EFeeds.addAllKeys(Collection<EMessageKeys>)`. These methods are needed to initially place message keys into the dictionary. But these methods are slow.

Once the message dictionary is filled out, it can be stored to persistent memory and restored quickly using the following methods:

- `EFeed.storeKeys(ObjectOutputStream)` Stores the message key dictionary to the given object output stream. The caller is responsible for correctly opening and closing the stream.
- `EFeed.storeKeys(Class<? extends EMessage>, ObjectOutputStream)` Stores only those message keys pertaining to a message class to the object output stream.
- `EFeed.storeKeys(Class<? extends EMessage>, Pattern, ObjectOutputStream)` Stores only those message keys which match the given message class and subject regular expression.

Once message key dictionary entries are persistent stored, these entries may be restored with the method `EFeed.loadKeys(ObjectInputStream).`

Message key dictionary entries can be retrieved using the following methods:

- `EFeed.findKeys()` Returns all message key dictionary entries.
- `EFeed.findKeys(Class<? extends EMessage>)` Returns only those message key dictionary entries pertaining to the given message class.
- `EFeed.findKeys(Class<? extends EMessage>, Pattern)` Returns only those message key dictionary entries pertaining to the given message class and whose subjects match the regular expression pattern.

The reason why a message key dictionary API is added to eBus is to support multi-subject feeds.

Feed me, Seymour!

Applications interface with eBus through feeds. So far only simple eBus feeds were used. A simple feed 1) contains a single message key, 2) connects a subscriber/requestor to one or more publishers/repliers, and 3) messages are exchanged only when sent while this connection is active.

eBus v. 4.5.0 begins the process of introducing complex feeds. The first of these new feeds are multi-subject feeds.

Multi-Subject Feeds

Multi-subject feeds act as a proxy between the application and multiple subordinate simple feeds. The multi-subject feed is responsible for configuring the subordinate feeds in the same way and keeping them in the same state: opened, advertised/subscribed, and closed. If new subordinate feeds are added, these new feeds are brought into the same configuration and state as the existing feeds.

But it is the subordinate simple feeds which interact with the application object. The multi-subject feed opens a subordinate simple feed, passing in the application object as the feed client. That means the subordinate feed calls back to the application object. If the application object creates a multi-subject feed with 1,000 subordinate feeds, then the application object receives callbacks from 1,000 subordinate feeds.

Multi-subject feeds behave in a similar manner to simple feeds but are not a `EFeed` subclass like simple feeds. They are opened, may have callbacks configured, advertised/subscribed, un-advertised/un-subscribed, and closed. Note the multi-subject feed configures the callback methods for the subordinate feeds based on how the multi-subject feed is configured. This means that each subordinate feed calls back to the same method. It is not possible for subordinate feeds belonging to the same multi-subject feed to call back to different methods.

There are four multi-subject feed types, matching the four simple feeds: `EMultiPublishFeed`, `EMultiSubscribeFeed`, `EMultiRequestFeed`, and `EMultiReplyFeed`. The multi-subject feeds also use the same role interfaces:

Multi-Subject Feed	Role
`EMultiPublishFeed`	`EPublisher`
`EMultiSubscribeFeed`	`ESubscriber`
`EMultiRequestFeed`	`ERequestor`
`EMultiReplyFeed`	`EReplier`

Subjects are defined in two ways: using a static `List<String>` containing the multiple subjects or a dynamic `net.sf.eBus.util.regex.Pattern` subject query. When a `List` is provided, subordinate feeds are created for each of the listed subjects. When `Pattern` is provided, the query is matched against existing subjects *for the multi-subject feed's class*. A subordinate feed is created for each of the matching subjects. The reason the subject query is dynamic is that when a new subject is created for the multi-subject feed's class and that new subject matches the query, a new subordinate feed is created for that subject with the subordinate feed move to the same state as the existing subordinate feeds.

Once a multi-subject feed is open, new subjects can be added or removed. The following methods may be used to add subjects:

- `addFeed(String subject)`: add a single subject to multi-subject feed.
- `addAllFeeds(List<String> subjects)`: add all listed subjects to multi-subject feed.
- `addAllFeeds(final Pattern query)`: add all existing subjects matching given pattern to multi-subject feed.

In each of the above methods, a subject is added only if it is not already in the feed. Conversely, existing feeds may be removed and closed using these methods:

- `closeFeed(String subject)`: close subordinate feed with the given subject.
- `closeAllFeeds(List<String> subjects)`: close all subordinate feeds for the given subjects.
- `closeAllFeeds(Pattern query)`: close all subordinate feeds with subjects matching the given query.

Since multi-subject feeds behave like their simple feed equivalents, the following code sample shows how to use an `EMultiReplyFeed`. This includes opening, configuring callback methods using lambda expressions, advertise, handle requests, add new subordinate feeds, and close the multi-subject reply feed. Please note the similarity between the `EReplyFeed` and `EMultiReplyFeed` code.

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

public class CatalogReplier
    implements EReplier
{
    private final List<String> mSubjects;
    private final FeedScope mScope;
    private EMultiReplyFeed mFeed;
    private final List<EReplyFeed.ERequest> mRequests;

    public CatalogReplier(final List<String> subjects, final FeedScope scope) {
        mSubjects = subjects;
        mScope = scope;
        mFeed = null;
        mRequests = new ArrayList<>();
    }

    @Override public void startup() {
        try {
            // This advertisement has no associated ECondition.
            // Use private methods to handle requests and cancels.
            mFeed =
                (EMultiReplyFeed.builder()).target(this)
                                           .scope(mScope)
                                           .messageClass(com.acme.CatalogOrder.class)
                                           .subjects(mSubjects)
                                           .requestCallback(this::handleRequest)
                                           .cancelRequestCallback(this::handleCancel)
                                           .build();
            mFeed.advertise();
            mFeed.updateFeedState(EFeedState.UP);
        } catch (Exception jex) {
            // Advertisement failed. Place recovery code here.
        }
    }

    private void handleRequest(final EReplyFeed.ERequest request,
                               final IEReplyFeed 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.
        }
    }

    private void handleCancel(final EReplyFeed.ERequest request,
                              final IEReplyFeed feed) {
        // Is this request still active? It is if the request is listed.
        if (mRequests.remove(request)) {
            // 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) && request.isActive()) {
            request.reply(new OrderReply(status, reason), ad);
            // If the request processing is complete, remove the request.
            if (status.isFinal()) {
                mRequests.remove(request);
            }
        }
    }

    // Adds a new subordinate feed for the given subject.
    public void openSubject(final String subject) {
        mFeed.addFeed(subject);
    }

    // Closes an existing feed for the given subject.
    public void closeSubject(final String subject) {
        mFeed.closeFeed(subject);
    }

    @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;
        }
    }
}
```

Pattern Feed

eBus v. 4.6.0 introduces pattern feeds. A pattern feed watches one or more notification feeds and reports a `MatchEvent` when events arriving on the notification feed(s) match the given pattern. The best way to explain how this works is through example.

A stock market algorithm reacts to the following trade pattern occurs in stock ABCD:

1. The pattern starts with a order quantity > 1,000 shares or the currently lowest priced trade.
2. Followed by 4 or more trades that is at a price > the average price of all previous trades (since this pattern match began). The quantity may increase or slowing decrease.
3. The pattern ends when the traded quantity is < 0.8 x the previously traded quantity.
4. The time difference between the first and last trade must ≤ 1 hour.
5. Events which are used for one match may *not* be used in another match. In other words, there is no intersection between event sets for any two matches *from the same patternl*. Exclusivity does not hold across patterns.

An eBus pattern is created using an `EventPattern.Builder`. A pattern consists of two parts: 1) notification feed(s) (known as pattern parameters or just parameters) and 2) the pattern. Parameters may be either built or provided to the pattern builder as a `java.util.Map`. The following example builds the parameters. A later example shows how to create a pattern parameter map.

Event patterns come in two flavors: ordered and unordered. This example is ordered since the events must arrive in a specified order. An unordered example is provided here.

Ordered Pattern

Building an event pattern follows these five steps:

1. Open the pattern builder with the pattern name, type (ordered or unordered), and (optional) parameter map.
2. Define the pattern parameters if a parameter map was not provided.
3. Define the pattern itself.
4. Define the optional until condition and whether this is an exclusive pattern.
5. Build the event pattern.

```
import net.sf.eBus.client.EventPattern;
import net.sf.eBus.client.EventPattern.PatternType;
import net.sf.eBus.messages.EMessageKey;

final String ticker = "ABCD";
final String orderParam = "ord";
final String tradeParam = "trd";
final String group = "g0";
final EventPattern.Builder builder =
    EventPattern.builder("ABCDTradeBlip", PatternType.ORDERED);                1
final EventPattern tradeBlip =
    builder.beginParameterMap()                                               2
           .beginParameter(orderParam)                                        3
           .messageKey(new EMessageKey(OrderMessage.class, ticker))           4
           .scope(FeedScope.REMOTE_ONLY)                                      5
           .endParameter()                                                    6
           .beginParameter(tradeParam) // Define the "trd" parameter second.
           .messageKey(new EMessageKey(TradeMessage.class, ticker))
           .scope(FeedScope.REMOTE_ONLY)
           .endParameter()
           .endParameterMap()                                                 7
           // Define the event pattern next.
           // Match 0: order quantity > 1,000 shares OR low trade.
           .beginMultiComponent()                                             8
           .matchCount(1, 1)                                                  9
           .addSubordinate(orderParam,                                        10
                        (e, g, u) -> {                                        11
                            final OrderMessage ord = (OrderMessage) e;
                            final boolean retcode = (ord.size > 1_000);

                            // If this event passes the test, then start adding up
                            // the trade prices to calculate the average price.
                            if (retcode) {
                                u.put("sum", ord.price);                      12
                                u.put("count", BigDecimal.ONE);
                            }

                            return (retcode);                                 13
                        })
           .addSubordinate(tradeParam,
                        (e, g, u) -> {
                            final TradeMessage trd = (TradeMessage) e;
                            final boolean retcode =
                                (trd.tradeType == PriceType.LOW)

                            if (retcode) {
                                u.put("sum", ord.price);
                                u.put("count", BigDecimal.ONE);
                            }

                            return (retcode);
                        })
```

```
.endMultiComponent() // Match 0: multi-component defined.      14
// Match 1: 4 or more trades with increasing price.
.beginGroup(group)                                              15
.beginSingleComponent(tradeParam)                               16
.matchCount(4, Integer.MAX_VALUE)                               17
.matchCondition((e, g, u) -> {
                final TradeMessage trd = (TradeMessage) e;
                final BigDecimal trdPx = (trd.trade).price;
                final BigDecimal pxSum = (BigDecimal) u.get("sum");
                final BigDecimal numTrades = (BigDecimal) u.get("count");
                final BigDecimal avgPx =                        18
                    pxSum.divide(numTrades, scale, RoundingMode.DOWN);
                final boolean retcode = (trdPx.compareTo(avgPx) > 0);

                // If this trade met the condition, then add price to sum.
                if (retcode) {                                  19
                    u.put("sum", pxSum.add(trdPx));
                    u.put("count", numTrades.add(BigDecimal.ONE));
                }

                return (retcode);
            })
.endSingleComponent()                                           20
.endGroup(group)                                                21
// Match 2: trade quantity drops.
.beginSingleComponent(tradeParam)
.matchCondition((e, g, u) -> {
                final int qty = (((TradeMessage) e).trade).size;
                final List<ENotificationMessage> trades =
                     g.get(EventPattern.ALL_EVENTS);
                final TradeMessage latest =
                     (TradeMessage) trades.get(trades.size() - 1);
                final int dropQty = (int) (0.8 * (latest.trade).size);

                return (qty < dropQty);
             })
.endSingleComponent()
.until((t, e) -> {                                              22
        final boolean retcode;

                if (t.isEmpty()) {
                    retcode = true;
                } else {
                    final TradeMessage first = (TradeMessage) t.get(0);
                    final long duration = (e.timestamp - first.timestamp);

                    retcode = (duration <= COMPLEX_TIME_LIMIT);
                }

                return (retcode);
            })
.isExclusive(true)                                              23
.build();                                                       24
```

1. Create an event pattern builder first. The first argument is the pattern name which is used as the `MatchEvent` subject. The second argument is the pattern type (ordered or unordered).
2. `builder.beginParameterMap()` starts the event pattern parameter definition(s).
3. `builder.beginParameter(String)` starts defining a single event parameter.
4. `builder.messageKey(EMessageKey)` defines the parameter *notification* message key. This is required for all parameters.
5. `builder.scope(EFeed.FeedScope)` defines the message feed scope. This is required for all parameters.
6. `builder.endParameter()` signals the end of the current parameter definition.
7. `builder.endParameterMap()` signals the end of all parameter definitions.
8. `builder.beginMultiComponent()` plays the same role as `java.util.regex.Pattern` character classes: it allows one of several events to appear at this point in the pattern match.
9. `builder.matchCount(n, m)` applies to the entire multi-component and may not be specified subordinate single events. Note that `builder.matchCount(1, 1)` is the default and did not need to be specified.
10. `builder.addSubordinate(param, condition)` adds a new single event to the multi-component. Please note that a parameter may appear multiple times as a subordinate as long as each appearance has a different condition.
    `builder.addSubordinate(param)` may be used to add a subordinate multi-component event without a condition.
11. `MatchCondition.test()` method interface has three parameters: `(e, g, u)` where:
    `e` is the latest event received which is to be tested for acceptability.
    `g` is the capturing groups map (capturing groups are explained in note **15**). This map does *not* contain `e` since `g` contains only successfully matched events and `e` is not yet matched.
    u is the user cache map defined as `Map<Object, Object>`. This is explained in the next point.
12. eBus event patterns allow the user to store or cache its own information associated with a given match instance. This cache is carried along a particular match and provided to `MatchCondition.test()` implementations and in the ultimate `MatchEvent` if the pattern is successfully matched. If the match fails, then the user cache instance is lost. Note that this user cache is unique to a given match and may not be shared between different matchings.
    eBus makes no limitations on what objects may be used as map keys or values. That is left entirely up to the user.
    In this example the user is storing a running sum of trade prices and trade count which are used later to calculate the average trade price - which is needed in later match condition tests.
13. Returning `true` means that event `e` is accepted and the match may continue. Returning `false` causes this match instance to fail and be removed from further consideration.
14. The multi-event component is now defined.
15. Users may defined named capturing groups in eBus patterns. The events which are accepted between `builder.beginGroup(name)` and `builder.endGroup(name)` are stored in the group map under the key `name` in the `List<ENotificationMessage>` value. Events are stored in acceptance order.

16. The second event component matches against a single event parameter **"trd"**.
17. `builder.matchCount(1, Integer.MAX_VALUE)` states the event must appear at least four times in a row to match the pattern.
18. The match condition retrieves the price summation and matched event count from the user cache in order to calculate the average trade price.
19. If event `e` satisfies the condition, then price summation and match event count updates are placed back into the user cache.
20. `builder.endSingleComponent()` closes off the single event component definition.
21. `builder.endGroup(group)` closes the named capturing group. Note that the group name needs is provided because capturing groups may be interleaved. This means that capturing groups may be opened and closed as follows: `beginGroup("g0")`, `beginGroup("g1")`, `endGroup("g0")`, `endGroup("g1")`.
22. `builder.until(BiPredicate<List<ENotificationMessage>, ENotificationMessage>)` condition must return `true` for the pattern to continue. The condition takes two parameters: matched events list and latest, unmatched event. The idea is that the pattern is valid only if an overall condition continues to hold. Often this condition checks if the first matched event and the latest event are within a fixed time frame but the check is not limited to duration.
23. `builder.isExclusive(true)` means that events in a successful match may not be used in another match *for the same pattern*. When an exclusive pattern matches, then all other in progress matches are disposed. Exclusivity does *not* apply across patterns. This means that an event may be used in multiple matches across multiple patterns.
24. `builder.build()` generates the `EventPattern` instance based on the previous configuration.

Unordered Pattern

The next example shows how to define an unordered pattern. This pattern specifies how many events of each type must be collected to match the pattern. These events may arrive in any order. The only thing that matters is the number of events.

The following looks for at 5 consecutive orders and trades demonstrating a price rise and there must be at least one order and one trade. This example uses the same parameter map as used in the ordered pattern example.

Note: this example assumes the same parameters as the ordered pattern example. Assume the pattern parameters are already defined.

```
builder.beginSingleComponent("ord")
        .matchCount(1, 4)                                                     1
        .endSingleComponent()
        .beginSingleComponent("trd")
        .matchCount(1, 4)                                                     1
        .endSingleComponent()
        .until((t, e) -> (t.size() < 6))                                      2
        .patternCondition(                                                    3
             p -> {
                 final List<ENotificationMessage> events =                    4
                     p.group(EventPattern.ALL_EVENTS);
                 // containsClass() returns true if the list contains at least one
                 // instance of the class.
                 final boolean hasOrder = containsClass(OrderReport.class, events);
                 final boolean hasTrade = containsClass(EquityTrade.class, events);
                 return (events.size() == 5 && hasOrder && hasTrade);
             }
        .build();                                                             5
```

1. There must be between and four (inclusive) order or trade events. This allows the fifth event to be the other event type.
2. The pattern look for exactly five events. The pattern is invalid on the sixth event.
3. `builder.patternCondition(Predicate<MatchEvent>)` defines the final pattern check before delivering the `MatchEvent` to the subscriber. If `Predicate.test(MatchEvent)` returns `true`, the match event is delivered; otherwise the match is dropped and the pattern continues searching.
4. `MatchEvent.group(String groupName)` returns the named capturing group's `List<ENotificationMessage>`. Even if a pattern has no explicitly defined capturing groups, the capturing group map always has an entry `EventPattern.ALL_EVENTS` which contains all matched events from first to last in acceptance order.
5. `builder.build()` returns the EventPattern based on the above configuration. Note that this pattern is *not* exclusive. This allows an event to satisfy multiple matching in the same pattern.

Defining Parameter Maps

If multiple event parameters are based on the same parameters, then defining a parameter map once for all patterns is desirable. The map definition is `Map<String, EventPattern.FeedInfo>` where the `String` key is the parameter name used in the preceding example and the `FeedInfo` defines the `ESubscribeFeed`. The parameter map definition is similar to the previous `EventPattern.Builder` example:

```
// All parameters use the new word feed.
final EMessageKey orderKey = new EMessageKey(OrderReport.class, "ACME");
final EMessageKey tradeKey = new EMessageKey(EquityTrade.class, "ACME");
final Map<String, EventPattern.FeedInfo> params = new HashMap<>();

// Defining "ord" parameter.
params.put("ord",                                                        1
            new EventPattern.FeedInfo(orderKey,                          2
                                      EFeed.FeedScope.REMOTE_ONLY));     3
                                                                         4

// Defining "trd" parameter.
params.put("trd",
           new EventPattern.FeedInfo(tradeKey,
                                      EFeed.FeedScope.REMOTE_ONLY));

// Create builder using the parameter map. The precludes the need for
// builder.beginParameterMap(), builder.endParameterMap().
final EventPattern.Builder builder =
   EventPattern.builder(“Trade Blip", EventPattern.PatternType.ORDERED, params);
```

1. Pattern map key is the parameter name string.
2. Define `ESubscribeFeed`’s `EMessageKey` (required).
3. Define `ESubscribeFeed`’s `EFeed.FeedScope` (required).
4. An `ESubscribeFeed ECondition` may be defined but is not provided in this example.

Subscribing to an Event Pattern Feed

Once the `EventPattern` is defined, the `EPatternFeed` can be opened and handled just like an `ESubscribeFeed`. The following is the previous `ESubscriber` example updated to work with a `EPatternFeed`.

```
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EFeedState;
import net.sf.eBus.client.EPatternFeed;
import net.sf.eBus.client.ESubscriber;
import net.sf.eBus.client.EventPattern;
import net.sf.eBus.client.IESubscribeFeed;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ENotificationMessage;

public class TradeBlipSubscriber
    implements ESubscriber
{
    // Pattern for the feed.
    private final EventPattern mPattern;

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

    // ABCDTradeBlip ordered pattern created by caller and passed in.
    public CatalogSubscriber(final EventPattern pattern) {
        mPattern = pattern;
        mFeed = null;
    }

    @Override public void startup() {
        try {
            // Pattern feed only needs two parameters.
            mFeed = (EPatternFeed.builder()).target(this)
                                            .eventPattern(mPattern)
                                            .build();
            mFeed.subscribe();
        } catch(IllegalArgumentException argex) {
            // Feed open failed. Place recovery code here.
        }
    }

    @Override public void feedStatus(final EFeedState feedState,
                                     final IESubscribeFeed 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 IESubscribeFeed feed) {
        msg is a net.sf.eBus.client.MatchEvent instance. Contains a trade blip match.
    }

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

Feed Interfaces

eBus v. 4.5.2 introduces four interfaces: `IEPublishFeed`, `IESubscribeFeed`, `IERequestFeed`, and `IEReplyFeed`. These interfaces contain method declarations common to the single- and multi-subject feeds. For example, `IESubscribeFeed` defines:

```
void subscribe()
void unsubscribe()
void statusCallback(FeedStatusCallback<ESubscribeFeed>)
void notifyCallback(NotifyCallback)
```

Since `IESubscribeFeed` extends `IEFeed`, the following method declarations are also included:

```
EFeed.FeedScope scope()
boolean isActive()
boolean inPlace()
boolean isFeedUp()
void close()
```

The single- and multi-subject feeds now implement their corresponding interface. This allows an application to store a feed reference using the interface. This is most helpful when an eBus client opens a mix of single- and multi-subject subscriptions. These subscription feeds can stored in a `List<IESubscribeFeed>`.

Don't Know Much About History[9]

eBus v. 6.3.0 introduces the `net.sf.eBus.feed.historic` package containing `EHistoricPublishFeed` and `EHistoricSubscribeFeed`. An historic publish feed works for an `IEHistoricPublisher` instance and uses an `IEMessageStore` instance to both persist and retrieve notification messages for a given type+topic message key. An historic subscribe feed retrieve past and/or live notification messages for an `IEHistoricSubscriber`.

The historic publish and subscribe feeds are examples of eBus hybrid objects. The historic publish feed is owned by an `IEHistoricPublisher` instance and runs in that owner's dispatcher. In turn the historic publish feed implements the `EPublisher` and `EReplier` interfaces. The `EPublisher` implementation is for posting live notification message to subscribers. The `EReplier` implementation is for responding to `HistoricRequest` messages. The historic notification message request contains an `net.sf.eBusx.time.EInterval` time interval. The historic publish feed retrieves those notification messages matching the time interval and sends them back in one or more `HistoricReply` messages (an historic reply contains a fixed number of notifications).

The historic subscribe feed is owned by an `IEHistoricSubscriber` instance and runs in that owner's dispatcher. In turn the historic subscribe feed implements the `ESubscriber` and `ERequestor` interfaces. The `ESubscriber` implementation is for receiving live notification messages. The `ERequestor` implementation is for requesting past notification messages.

Note that the historic publish and subscribe feeds do *not* implement the generic `IEPublishFeed` or `IESubscribeFeed` interfaces due to the fact they are hybrid objects which use the generic eBus interfaces.

Note that `IESubscribeFeed` may access live notification messages published by an `EHistoricPublishFeed`. Conversely `EHistoricSubscribeFeed` may access live notification messages published by an `IEPublishFeed`.

The steps for using historic feeds are shown below.

[9] From the lyrics of "Wonderful World" by Sam Cooke, Lou Adler, and Herb Alpert

IEHistoricPublisher

Step 1: Implementing an historic publisher

An application can publish messages which are both published live (if there are any active subscribers) and persisted to a message store by first implementing the `net.sf.eBus.feed.historic.IEHistoricPublisher` interface.

Note: historic feeds are dependent on publishers being given a unique 8-byte identifier in order to distinguish between notification message stream sources.

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

```
import net.sf.eBus.feed.historic.IEHistoricPublisher;

public class CatalogPublisher
    implements IEHistoricPublisher
{
    // Unique publisher identifier used to distinguish this publisher within the
    // application.
    private final long mPublisherId;

    // Returns unique publisher identifier. This identifier must be placed in all
    // outbound notification messages.
    @Override public long publisherId() {
        return (mPublisherId);
    }

    // This callback is that same as in EPublisher and is provided as a courtesy only
    // since the underlying historic publish feed tracks when a live message may be
    // published.
    // Default interface implementation does nothing. Therefore this implementation is
    // not needed.
    @Override public void publishStatus(final EFeedState feedState,
                                        final EHistoricPublishFeed feed)
    {}
}
```

Step 2: Implementing a message store

This class is used to persist an outbound notification message and retrieve stored notification messages upon an historic subscribe feed's request. This message store needs to store both the target notification message and `net.sf.eBus.feed.historic.PublishStatusEvent` notification message. `PublishStatusEvent` tracks when an historic publisher's notification feed transitions between being down (or unknown) to up. This allows a subscriber to determine when there is a break in the publisher's notification message stream.

This manual does not cover how a message store may be implemented. It could use an OS flat file, relational database, NOSQL store, time series database, etc. It is also up to the implementation on how to store a notification message in the target persistent store.

Note: a message store instance may be shared among multiple historic notification publish feeds. But the message store implementation is responsible for providing thread-safe persistence and retrieval.

See **Message Store** for two message store implementations provided in eBus release 7.2.

```
import net.sf.eBus.feed.historic.IEMessageStore;

public class CatalogStore
    implements IEMessageStore
{
    // Returns true if this message store is open and false otherwise.
    @Override public boolean isOpen() {
        …
    }

    // Persists given message to store. May be target message or PublishStatusEvent.
    @Override public void store(final ENotificationMessage msg) {
        …
    }

    // Returns collection containing both target notification and PublishStatusEvent
    // messages covering the given time interval.
    @Override public Collection<ENotificationMessage> retrieve(final EInterval iv) {
        …
    }
}
```

Step 3: Opening, starting, and advertising an historic publish feed

Historic publish feeds are opened using `EHistoricPublishFeed.Builder instance`. This builder instance is acquired using the static method `EHistoricPublishFeed.builder(EMessageKey key, IEHistoricPublisher publisher)` where `key` is the notification type+topic message key and `publisher` is the historic publisher instance. This builder is then used to create an open historic publish feed:

```
final EHistoricPublishFeed.Builder builder =
    EHistoricPublishFeed.builder(key, publisher);

pubFeed = builder.name(publisher name)
                 .scope(scope)
                 .messageStore(message store)
                 .notificationsPerReply(num messages)
                 .build();
```

where:

`publisher name` is an optional name placed into feed log messages.

`scope` is the feed scope (local only, remote only, or local+remote).

`message store` is used to persist and retrieve notification messages.

`num messages` is the maximum number of retrieved notification messages placed in an `HistoricReply` message. This means that an historic publisher may need to post multiple replies to cover all retrieved notifications.

The second step is to start up the historic publish feed. This is necessary since `EHistoricPublishFeed` is itself an `EObject` instance. Since the historic publish feed operates within the historic publisher's dispatcher, this is acceptable:

```
pubFeed.startup();
```

The third step is to advertise your `EHistoricPublishFeed` to historic subscribers:

```
pubFeed.advertise();
```

The fourth step is to inform historic subscribers that the feed is up:

```
pubFeed.updateFeedState(EFeedState.UP);
```

The historic publish feed state *must* be declared as up prior to publishing any target notification messages. Failure to do so results in `EHistoricPublishFeed.publish(ENotificationMessage)` throwing an `IllegalStateException`.

```java
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EFeedState;
import net.sf.eBus.feed.historic.EHistoricPublishFeed;
import net.sf.eBus.feed.historic.IEHistoricPublisher;
import net.sf.eBus.messages.EMessageKey;

public class CatalogPublisher
    implements IEHistoricPublisher
{
    // Unique publisher identifier.
    private final long mPublisherId;

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

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

    // Persistent message store.
    private final IEMessageStore mStore;

    // Advertise and publish on this feed.
    private EHistoricPublishFeed mFeed;

    // Current message position.
    private int mPosition;

    public CatalogPublisher(final long pubId,
                            final String subject,
                            final FeedScope scope,
                            final IEMessageStore ms) {
        mPublisherId = pubId;
        mKey = new EMessageKey(CatalogUpdate.class, subject);
        mScope = scope;
        mStore = ms;
    }

    @Override public void startup() {
        try {
            final EHistoricPublishFeed.Builder builder =
                EHistoricPublishFeed.builder(mKey, this);

            mFeed = builder.scope(mScope)
                           .messageStore(mStore)
                           .notificationsPerReply(20)
                           .build();
            mFeed.startup();
            mFeed.advertise();
            mFeed.updateFeedState(EFeedState.UP);
        } catch (Exception jex) {
            // Failed to start historic publish feed. Place recovery code here.
        }
    }

    @Override public void shutdown() {
        Shown in step 5.
    }
}
```

Step 4: Publishing

Once an historic publish feed is successfully opened, started, advertised, and feed state is set to up, an historic publisher is free to start posting notification messages. The historic publisher does not need to wait for active subscribers to arrive. This allows notification messages to be persisted in the absence of subscribers.

Publishing notification messages is the same as with an `IEPublishFeed` *except*:

- The notification message publish identifier must be set and must match the value returned by `EHistoricPublisher.publisherId().`
- Publisher is strongly encouraged to set notification message position value since historic notification messages are sorted by `ENotificationMessage.timestamp`, `ENotificationMessage.publisherId`, and `ENotificationMessage.position` before delivery to historic subscriber. Since the message `timestamp` field has millisecond granularity, if the publisher posts multiple messages per millisecond, the position is needed to properly order historic messages.

```
import java.time.Instant;

public class CatalogPublisher
    implements IEHistoricPublisher
{
    // Roll over message position to zero when position reaches 1,000.
    private static final int MAX_POSITION = 1_000;

    public void updateProduct(final String productName,
                              final Money price,
                              final int stockQty) {
        // Is the feed open and in place?
        if (mFeed != null && mFeed.isFeedUp()) {
            // Yes, clear to send a catalog update.
            mFeed.publish((CatalogUpdate.builder()).subject(mKey.subject())
                                                   .timestamp(Instant.now())
                                                   .publishId(mPublisherId)
                                                   .position(mPosition)
                                                   .productName(productName)
                                                   .price(price)
                                                   .stockQty(stockQty)
                                                   .build());

            ++mPosition;
            if (mPosition == MAX_POSITION) {
                mPosition = 0;
            }
        }
    }
}
```

Step 5: Unadvertising the publisher

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

1. `EHistoricPublishFeed.updateFeedState(EFeedState.DOWN)`: This tells eBus that the publisher is *temporarily* unable to publish notifications on this feed but may resume in the future (after setting the feed state to up). The feed remains advertised and available to subscribers. The historic publish feed sends a `PublishStatusEvent` with the publisher identifier and feed state set appropriately.
2. `EHistoricPublishFeed.unadvertise()`: Publisher is retracting the live notification feed and subscribers will no long receive live notifications from this historic publisher. The historic notification reply feed remains advertised and historic subscribers can still retrieve past notifications. The historic publisher may advertise the live notification feed again and does not need to open a new feed. Historic subscribers are informed that the publisher state is unknown.
3. `EHistoricPublishFeed.shutdown()`: Publisher is permanently closing the historic publish feed. Publisher feed state is set to unknown (if not already set to that value). Once closed, the historic publish feed may *not* be used again. If the historic publisher intends to publish again, then a new historic publish feed must be opened and used.

Note: Historic publish feeds maintain a *strong* reference to its historic publisher. Since an historic publisher should not share its historic publish feed reference with objects outside the publisher, the circular reference between historic publisher and historic publish feed will not prevent garbage collection of these objects.

```
public class CatalogPublisher
    implements IEHistoricPublisher
{
    // Retract historic notification feed.
    if (mFeed != null) {
        // unadvertise() unnecessary since shutdown() retracts in-place advertisement.
        mFeed.shutdown();
        mFeed = null;
    }
}
```

Step 6: Complete historic publisher code

```java
import java.time.Instant;
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EFeedState;
import net.sf.eBus.feed.historic.EHistoricPublishFeed;
import net.sf.eBus.feed.historic.IEHistoricPublisher;
import net.sf.eBus.messages.EMessageKey;

public class CatalogPublisher
    implements IEHistoricPublisher
{
    // Roll over message position to zero when position reaches 1,000.
    private static final int MAX_POSITION = 1_000;

    // Unique publisher identifier.
    private final long mPublisherId;

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

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

    // Persistent message store.
    private final IEMessageStore mStore;

    // Advertise and publish on this feed.
    private EHistoricPublishFeed mFeed;

    // Current message position.
    private int mPosition;

    public CatalogPublisher(final long pubId,
                            final String subject,
                            final FeedScope scope,
                            final IEMessageStore ms) {
        mPublisherId = pubId;
        mKey = new EMessageKey(CatalogUpdate.class, subject);
        mScope = scope;
        mStore = ms;
    }

    @Override public void startup() {
        try {
            final EHistoricPublishFeed.Builder builder =
                EHistoricPublishFeed.builder(mKey, this);

            mFeed = builder.scope(mScope)
                           .messageStore(mStore)
                           .notificationsPerReply(20)
                           .build();
            mFeed.startup();
            mFeed.advertise();
            mFeed.updateFeedState(EFeedState.UP);
        } catch (Exception jex) {
            // Failed to start historic publish feed. Place recovery code here.
        }
    }

    @Override public void shutdown() {
        if (mFeed != null) {
            mFeed.close();
            mFeed = null;
        }
    }

    public void updateProduct(final String productName,
                              final Money price,
                              final int stockQty) {
        // Is the feed open and in place?
        if (mFeed != null && mFeed.isFeedUp()) {
            // Yes, clear to send a catalog update.
            mFeed.publish((CatalogUpdate.builder()).subject(mKey.subject())
                                                    .timestamp(Instant.now())
                                                    .publishId(mPublisherId)
                                                    .position(mPosition)
                                                    .productName(productName)
                                                    .price(price)
                                                    .stockQty(stockQty)
                                                    .build());

            ++mPosition;
            if (mPosition == MAX_POSITION) {
                mPosition = 0;
            }
        }
    }
}
```

IEHistoricSubscriber

Step 1: Implementing an historic subscriber

Every application wishing to place historic notification subscriptions must implement `net.sf.eBus.feed.historic.IESubscriber` interface.

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

```
import net.sf.eBus.feed.historic.EHistoricSubscribeFeed;
import net.sf.eBus.feed.historic.EHistoricSubscribeFeed.HistoricFeedState;
import net.sf.eBus.messages.ENotificationMessage;

public class CatalogSubscriber
    implements IEHistoricSubscriber
{
    @Override public void feedDone(final HistoricFeedState feedState,
                                   final EHistoricSubscribeFeed feed) {
        See step 5.
    }

    @Override public void feedStatus(final PublishStatusEvent event,
                                     final EHistoricSubscribeFeed feed) {
        See step 3.
    }

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

Step 2: Opening, starting and subscribing historic feed

Historic subscribe feed allows an historic subscriber to access notification messages published in the past and on to a future time. This example retrieves past notification messages at an inclusive begin time and into the future to an exclusive end time.

The first step is opening the historic subscribe feed:

```
final EHistoricSubscribeFeed.Builder builder =
    EHistoricSubscribeFeed.builder(key, subscriber);

subFeed = builder.name(subscriber name)
                 .scope(scope)
                 .condition(condition)
                 .from(begin time, begin clusivity)
                 .to(end time, end clusivity)
                 .build();
```

where:

key is the message key containing the notification message key and subject.

subscriber is the `EHistoricSubscriber` instance owning the historic subscribe feed.

subscriber name is used for logging purposes only. This property is optional.

scope is the feed scope.

condition is the optional condition used to restrict delivered notification messages (both past and live) to those which satisfy the condition.

begin time *must* be a time in the past. If the historic subscribe feed should start with the current time, then call `builder.fromNow()`.

begin clusivity is set to either `EInterval.Clusivity.INCLUSIVE` or `EInterval.Clusivity.EXCLUSIVE`.

end time may be in the past or future. If in the past, then only historic notification messages are sent to the historic subscriber. If in the future, then the historic subscribe feed terminates when the end time is reached.

end clusivity is set to `EInterval.Clusivity.INCLUSIVE` or `EInterval.Clusivity.EXCLUSIVE`.

The second step starts the historic subscribe feed:

```
subFeed.startup();
```

The second step puts the subscription in place:

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

This begins the process of retrieving historic notification messages and receiving live messages.

Note: EHistoricSubscribeFeed attempts to guarantee that there are no redundant or missing notification messages in its feed stream, this result is not guaranteed.

```
import java.time.Instant;
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ENotificationMessage;
import net.sf.eBusx.time.EInterval.Clusivity;

public class CatalogSubscriber
    implements IEHistoricSubscriber
{
    // Notification message class and subject and feed scope.
    private final EMessageKey mKey;
    private final FeedScope mScope;

    // Historic feed begin and end times and clusivity.
    private final Instant mBeginTime;
    private final Clusivity mBeginClusivity;
    private final Instant mEndTime;
    private final Clusivity mEndClusivity;

    // Store feed reference here so it may be used to unsubscribe.
    private EHistoricSubscribeFeed mFeed;

    public CatalogSubscriber(final String subject,
                             final FeedScope scope,
                             final Instant beginTime,
                             final Clusivity beginClusivity,
                             final Instant endTime,
                             final Clusivity endClusivity) {
        mKey = new EMessageKey(CatalogUpdate.class, subject);
        mScope = scope;
        mBeginTime = beginTime;
        mBeginClusivity = beginClusivity;
        mEndTime = endTime;
        mEndClusivity = endClusivity;
    }

    @Override public void startup() {
        try {
            final EHistoricSubscribeFeed.Builder builder =
                EHistoricSubscribeFeed.builder(mKey, this);

            mFeed = builder.scope(mScope)
                           .from(mBeginTime, mBeginClusivity)
                           .to(mEndTime, mEndClusivity)
                           .build();
            mFeed.startup();
            mFeed.subscribe();
        } catch (Exception jex) {
            // Feed open failed. Place recovery code here.
        }
    }
}
```

Step 3: Handling publisher feed status

Historic feeds inform historic subscribers when an historic publisher's feed status changes using the `PublishStatusEvent`. This notification message contains:

- Unique publisher identifier. The importance of historic publishers being assigned an application-wide unique identifier cannot be stressed enough. It is needed so that historic subscribers can match publisher feed state with publisher notification message stream.
- Notification message key. This is the publisher feed stream message key and *not* `PublishStatusEvent` key.
- Latest publisher feed state.

This callback allows the historic subscriber the ability to determine if there is a break in an individual publisher's notification message stream.

IEHistoricSubscriber.feedStatus method may be replaced by another subscriber lambda expression using `EHistoricSubscribeFeed.Builder.statusCallback(HistoricFeedStatusCallback).`

```
import net.sf.eBus.feed.historic.PublishStatusEvent;

public class CatalogSubscriber
    implements IEHistoricSubscriber
{
    @Override public void feedStatus(final PublishStatusEvent event,
                                     final EHistoricSubscribeFeed feed) {
        if (event.feedState == EFeedState.UP) {
            // Mark event.publisherId as up.
        }
        else {
            // Mark event.publisherId as down.
        }
    }
}
```

Step 4: Handling notifications

Both past and/or live notification messages are delivered to to `IEHistoricSubscriber.notify` method.

This interface method may be replaced by a another subscriber lambda expression using `EHistoricSubscribeFeed.Builder.notifyCallback(HistoricNotifyCallback)`.

```
public class CatalogSubscriber
    implements IEHistoricSubscriber
{
    @Override public void notify(final ENotificationMessage msg,
                                 final EHistoricSubscribeFeed feed) {
        // Process notification message.
    }
}
```

Step 5: Handling historic feed completion

If the historic subscribe feed is set to end at a fixed time (past or present), then `IEHistoricSubscriber.feedDone(HistoricFeedState, EHistoricSubscribeFeed)` is called when the historic subscribe feed reaches that end time. If the historic subscribe successfully completed, then feed state is set to `HistoricFeedState.DONE_SUCCESS`. Otherwise feed state is `HistoricFeedState.DONE_ERROR` and `EHistoricSubscribeFeed.errorCause()` may be called to determine why the historic subscribe feed failed.

This interface method be be replaced by another subscriber lambda expression using `EHistoricSubscribeFeed.Builder.doneCallback(HistoricFeedDoneCallback).`

```
public class CatalogSubscriber
    implements IEHistoricSubscriber
{
    @Override public void feedDone(final HistoricFeedState feedState,
                                   final EHistoricSubscribeFeed feed) {
        if (feedState == HistoricFeedState.DONE_SUCCESS) {
            // Clean up after historic subscribe feed completion.
        }
        else if (feedState == HistoricFeedState.DONE_ERROR) {
            // Place error recovery code here.
        }
    }
}
```

Step 6: Retracting historic feed.

A subscriber has only one way to retract an historic subscription: `EHistoricSubscribeFeed.shutdown()`. This is because historic subscribe feeds have an associated begin and end time. Unsubscribing and re-subscribing an historic feed would only result in re-playing the same notification messages.

```
public class CatalogSubscriber
    implements IEHistoricSubscriber
{
    @Override public void shutdown() {
        if (mFeed != null) {
            mFeed.shutdown();
            mFeed = null;
        }
    }
}
```

Step 7: Complete historic subscriber code

```java
import java.time.Instant;
import net.sf.eBus.feed.historic.EHistoricSubscribeFeed;
import net.sf.eBus.feed.historic.EHistoricSubscribeFeed.HistoricFeedState;
import net.sf.eBus.feed.historic.PublishStatusEvent;
import net.sf.eBus.messages.ENotificationMessage;
import net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBus.messages.ENotificationMessage;
import net.sf.eBusx.time.EInterval.Clusivity;

public class CatalogSubscriber
    implements IEHistoricSubscriber
{
    // Notification message class and subject and feed scope.
    private final EMessageKey mKey;
    private final FeedScope mScope;

    // Historic feed begin and end times and clusivity.
    private final Instant mBeginTime;
    private final Clusivity mBeginClusivity;
    private final Instant mEndTime;
    private final Clusivity mEndClusivity;

    // Store feed reference here so it may be used to unsubscribe.
    private EHistoricSubscribeFeed mFeed;

    public CatalogSubscriber(final String subject,
                             final FeedScope scope,
                             final Instant beginTime,
                             final Clusivity beginClusivity,
                             final Instant endTime,
                             final Clusivity endClusivity) {
        mKey = new EMessageKey(CatalogUpdate.class, subject);
        mScope = scope;
        mBeginTime = beginTime;
        mBeginClusivity = beginClusivity;
        mEndTime = endTime;
        mEndClusivity = endClusivity;
    }

    @Override public void startup() {
        try {
            final EHistoricSubscribeFeed.Builder builder =
                EHistoricSubscribeFeed.builder(mKey, this);

            mFeed = builder.scope(mScope)
                           .from(mBeginTime, mBeginClusivity)
                           .to(mEndTime, mEndClusivity)
                           .build();
            mFeed.startup();
            mFeed.subscribe();
        } catch (Exception jex) {
            // Feed open failed. Place recovery code here.
        }
    }

    @Override public void shutdown() {
        if (mFeed != null) {
            mFeed.shutdown();
            mFeed = null;
        }
    }
    @Override public void feedStatus(final PublishStatusEvent event,
                                     final EHistoricSubscribeFeed feed) {
        if (event.feedState == EFeedState.UP) {
            // Mark event.publisherId as up.
        }
        else {
            // Mark event.publisherId as down.
        }
    }

    @Override public void notify(final ENotificationMessage msg,
                                 final EHistoricSubscribeFeed feed) {
        // Process notification message.
    }

    @Override public void feedDone(final HistoricFeedState feedState,
                                   final EHistoricSubscribeFeed feed) {
        if (feedState == HistoricFeedState.DONE_SUCCESS) {
            // Clean up after historic subscribe feed completion.
        }
        else if (feedState == HistoricFeedState.DONE_ERROR) {
            // Place error recovery code here.
        }
    }
}
```

Message Store

eBus v. 7.2.0 provides two message store implementations: in-memory and SQL. As noted in above, message stores store a specified target eBus notification message type (as well as `net.sf.eBus.feed.historic.PublishStatusEvent` as which report when an historic publisher's feed changes state). `EHistoricPublishFeed` guarantees that the correct target notification message type is stored. The eBus-provided message stores are designed to work only with historic feeds, they do not validate inputs as the historic feed guarantees argument correctness.

If these message stores are used outside of the eBus historic message feed framework, then the application is also responsible for guaranteeing that type correct, non-null messages are stored and non-null intervals are provided to message retrieval.

In-memory Message Store

`InMemoryMessageStore` uses a fixed-length array to store and retrieve target notification message types and `PublishStatusEvent`. This fixed-length array is treated as a ring buffer. When the in-memory store reaches its maximum allowed size, the oldest message is overwritten with the new message.

In-memory message store supports message retrieval based on an `EInterval` (as required by `IEMessageStore` interface) and the last *n* messages stored. The later is *not* supported by eBus historic feed but provided for application use and does validate the given message count argument is > zero.

Please note that this message store does *not* persist stored messages so when the store is closed, the messages are discarded.

SQL Message Store

`SqlMessageStore` provides a partial interface between `EHistoricPublishFeed` and a Java SQL connection (`java.sql.Connection`). An application using this message store is responsible for:

- Defining the store notification message key.
- Providing an `IInertGenerator` instance used to generate an SQL statement for storing an eBus notification message (as defined by the message key) or `PublishStatusEvent` into target database.
- Providing an `IRetrieveGenerator` instance used to generate SQL statement for retrieving eBus notification message or `PublishStatusEvent` from target database based on given message key and `EInterval`.
- Providing an `IMessageDecoder` instance used to translate a `ResultSet` into eBus notification message (either store's target message key or `PublishStatusEvent`).

Note: application is responsible for inserting, retrieving, and decoding both the message type defined by the message key *and* `PublishStatusEvent` messages. Also note that the notification type must be inserted and retrieved using a single SQL statement. Notification types too complex for this requirement cannot use `SqlMessageMessage`.

There is a one-to-one mapping between an `SqlMessageStore` and eBus notification message key. But the application developer is free to design SQL tables in any way deemed best. Most likely this means that there will be a single table for a given eBus notification message class containing the message subject. This allows a single `SqlMessageBus` instance to be share among multiple

EHistoricPublishFeeds. In this case, it is the application's responsibility to provide thread-safe message store and retrieval.

Simply put, the application is responsible for interfacing SqlMessageStore with the application database, translating eBus notification messages between database and Java.

The following code is used to insert and retrieve a TopOfBookMessage into and from an SQL table:

```
public final class TopOfBook extends ENotificationMessage {
    public final PriceSize bid;
    public final PriceSize ask;
    public final PriceType priceType; // enum (opening, closing, latest, etc.)
    …
}

public final class PriceSize extends EField {
    public final Decimal6f price; // Uses decimal4j for prices.
    public final int size;
    public final Trend trend; // enum (up, down in relation to previous message)
    …
}
```

The tables used to store TopOfBook and PriceSize types as well as enums are defined as follows (note: table indices are not provided, defined using Postgresql):

```
CREATE TYPE price_type AS ENUM (
    'LATEST',
    'OPEN',
    'CLOSE',
    'LOW',
    'HIGH'
);

CREATE TYPE trend AS ENUM (
    'NA',
    'UP',
    'DOWN',
    'ZERO_PLUS',
    'ZERO_MINUS'
};

CREATE TYPE price_size AS (
    price       NUMERIC(15,6) NOT NULL,
    size        integer NOT NULL,
    price_trend trend NOT NULL
);

CREATE TABLE top_of_book (
    subject            varchar(500) NOT NULL,
    message_timestamp  timestamp NOT NULL,
    publisher_id       bigint NOT NULL,
    publisher_position integer NOT NULL,
    bid                price_size NOT NULL,
    ask                price_size NOT NULL,
    price_type         price_type NOT NULL,
    PRIMARY KEY ( message_timestamp, publisher_id, publisher_position )
);
```

The generated SQL statement used to insert a TopOfBook message is:

```
INSERT INTO top_of_book VALUES ('ACME', '2024-01-06 07:30:53:113', 2001, 1, ROW
(12.76, 1200, 'ZERO_PLUS'::trend), ROW(12.77, 600, 'UP'::trend), 'LATEST'::price_type)
```

The generated SQL statement to retrieve a `TopOfBook` message for the `EInterval` [2024-01-06 07:47, 2024-01-06 07:50) is:

```
SELECT subject, message_timestamp, publisher_id, publisher_position, (bid).price,
(bid).size, (bid).price_trend, (ask).price, (ask).size, (ask).price_trend, price_type
FROM top_of_book WHERE message_timestamp >= '2024-01-06 07:47:00' AND
message_timestamp < '2024-01-06 07:50:00'
```

The `IMessageDecoder` method to translate a `ResultSet` to `TopOfBook` instance is:

```
@Override public ENotificationMessage toMessage(EMessageKey key, ResultSet rs)
    throws SQLException {
    final String tickerSymbol = rs.getString(1);
    final Instant timestamp = (rs.getTimestamp(2)).toInstant();
    final long publisherId = rs.getLong(3);
    final int position = rs.getInt(4);
    final PriceSize bid =
        (PriceSize.builder()).price(Decimal6f.valueOf(rs.getBigDecimal(5)))
                             .size(rs.getInt(6))
                             .trend(Trend.valueOf(rs.getString(7)))
                             .build();
    final PriceSize ask =
        (PriceSize.builder()).price(Decimal6f.valueOf(rs.getBigDecimal(8)))
                             .size(rs.getInt(9)
                             .trend(Trend.valueOf(rs.getString(10)))
                             .build();
    final PriceType priceType = PriceType.valueOf(rs.getString(11));
    final TopOfBookMessage.Builder builder = TopOfBookMessage.builder();

    return (builder.subject(tickerSymbol)
                   .timestamp(timestamp)
                   .publisherId(publisherId)
                   .position(position)
                   .bid(bid)
                   .ask(ask)
                   .priceType(priceType)
                   .build()));
}
```

The SQL and Java for `PublishStatusEvent` messages is not shown but is similar to `TopOfBook`.

Persisting Messages

eBus v. 6.2.0 adds the interface `net.sf.eBus.client.IMessageExhaust`. An application implements `IMessageExhaust` and passes an instance of that implementation to `EFeed.setExhaust(IMessageExhaust)`. The local eBus then passes *all* notification, request, and reply messages flowing through eBus to the registered message exhaust.

Note: only *one* exhaust instance may be may be registered at a time. If the application does not want to exhaust all messages, then the `IMessageExhaust` implementation is responsible for filtering out unwanted messages. If the application needs to exhaust messages to different persistent stores, again the registered exhaust is responsible for interfacing with those multiple stores.

The application is responsible for opening and closing exhaust persistent stores when appropriate. Message exhaust is "turned off" by passing a null `IMessageExhaust` value to `EFeed.setExhaust`. resulting in the default message exhaust which does nothing with the given eBus message. `IMessageExhaust.exhaust(EMessage)` method is called from an eBus dispatcher thread. This means that message exhaust does not interfere with eBus message forwarding - meaning that the exhaust process is not in-line with the message forwarding process.

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 must specify either 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.

eBus v. 4.7.0 introduces SSL/TLS **secure TCP connections**. Secure eBus service and connections *cannot* be defined in eBus properties as this requires storing sensitive security information in the clear. Therefore, eBus secure services and connections can only be created using the eBus `EConfigure` builder API. **Go here to learn how to do this**.

eBus v. 5.3.0 introduces unreliable UDP connections. No matter what connection type used, only one connection is allowed between eBus applications. While UDP is a connection-less protocol, eBus is connection-based. So an application wanting to receive UDP connections must open a UDP service port. This port, like a TCP service, creates a new UDP socket when it "accepts" a UDP connect request.

eBus v. 6.0.0 introduces DTLS **secure UDP connections**.

eBus v. 7.0.0 introduces reliable UDP connections (both in the clear and secure).

Both TCP and UDP connection types may appear in configuration files. Therefore a new property key `connectionType` has been added. This property is required.

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

Appendix B shows how eBus protocol used between eBus applications.

Please note: As of eBus v. 6.0.0 configuration using Java Properties is no longer supported. Property files must be converted to JSON typesafe HOCON syntax.

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: Services list.
Description: JSON typesafe array of services.
Type: JSON array of `EConfigure.Service` configurations.
Optional? Yes.
Check: None.
Default: No services.
Keyword: `services`

Name: Service name.
Description: Unique service name within services array.
Type: `String`
Optional? No.
Check: Is not empty string. Must be unique within service list. Service name has no formatting restrictions.
Default: No default.
Keyword: `name`

Name: Connection type.
Description: Either "TCP" or "UDP".
Type: `net.sf.eBus.config.EConfigure.ConnectionType`
Optional? Yes
Check: Must be a valid connection type name. May *not* be a secure connection type if defined in a configuration file.
Default: `ConnectionType.TCP`
Keyword: `connectionType`

Name: Port number.
Description: The TCP or UDP service port number.
Type: `int`
Optional? No.
Check: 0 ≤ port ≤ 65,536
Default: None.
Keyword: `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: Array of `net.sf.eBus.client.AddressFilter`
Optional? Yes.
Check: None.
Default: No filter, all client connections are accepted.
Keyword: `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: `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: `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: `int`
Optional? Yes.
Check: > zero.
Default: 2,048 bytes.
Keyword: `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: `int`
Optional? Yes.
Check: > zero.
Default: 2,048 bytes.
Keyword: `outputBufferSize`

Name: Buffer byte order
Description: Set the socket input and output buffer to this byte order.
Type: `java.nio.ByteOrder`
Optional? Yes.
Check: must be either `ByteOrder.BIG_ENDIAN` or `ByteOrder.LITTLE_ENDIAN`.
Default: `ByteOrder.LITTLE_ENDIAN`
Keyword: `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: `int`
Optional? Yes.
Check: ≥ zero.
Default: Zero (unlimited queue size).
Keyword: `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: `java.time.Duration`
Optional? Yes.
Check: ≥ zero.
Default: Zero (no heart-beating)
Keyword: `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: `java.time.Duration`
Optional? Yes.
Check: ≥ zero.
Default: Zero (wait indefinitely)
Keyword: `heartbeatReplyDelay`

Name: Can Pause Flag.
Description: Specifies whether accepted connections may *accept* pauses requests or not.
Type: `boolean`
Optional? Yes.
Check: None.
Default: false (accepted connections may not be paused).
Keyword: `canPause`

Name: Maximum allowed pause duration.
Description: Accepted connections are allowed to pause for this maximum allowed time.
Type: `java.time.Duration`
Optional? Yes but must be specified if `canPause` is `true`.
Check: ≥ zero.
Default: No default value if this property is required; zero if not required.
Keyword: `pauseTime`

Name: Maximum allowed message backlog size.
Description: Paused connections will backlog at most this many messages. Once the maximum is reached, messages will be discarded to keep the backlog size at the maximum.
Type: `int`
Optional? Yes but must be specified if `canPause` is `true`.
Check: > zero.
Default: No default value if this property is required; zero if not required.
Keyword: `maxBacklogSize`

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: Connections list.
Description: JSON typesafe array of remote connections.
Type: JSON array of `EConfigure.RemoteConnection` configurations.
Optional? Yes.
Check: None.
Default: No connections.
Keyword: `eBus.connections`

Name: Connection name.
Description: Unique connection name within connections array.
Type: `String`
Optional? No.
Check: Is not empty string. Must be unique within service list. Connection name has no formatting restrictions.
Default: No default.
Keyword: `name`

Name: Connection type.
Description: Either "TCP", "UDP", "SECURE_TCP", "SECURE_UDP", "RELIABLE_UDP", "SECURE_RELIABLE_UDP"
Type: `net.sf.eBus.config.EConfigure.ConnectionType`
Optional? Yes
Check: Must be a valid connection type name.
Default: `ConnectionType.TCP`
Keyword: `connectionType`

Name: Internet address and port.
Description: Remote eBus server Internet address.
Type: `String`
Optional? No.
Check: Not null or empty string. Must be a valid host name or dotted IP address notation.
Default: No default.
Keyword: `host`

Name: Internet port.
Description: Remote eBus service port.
Type: `int`
Optional? No.
Check: ≥ zero.
Default: No default.
Keyword: `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: `int`.
Optional? Yes.

Check: ≥ zero.
Default: `AsyncSocket.ANY_PORT` (local port assigned by the OS).
Keyword: `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: `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: `int`
Optional? Yes.
Check: > zero.
Default: 2,048 bytes.
Keyword: `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: `int`
Optional? Yes.
Check: > zero.
Default: 2,048 bytes.
Keyword: `outputBufferSize`

Name: Buffer byte order.
Description: Set the socket input and output buffer to this byte order.
Type: `java.nio.ByteOrder`
Optional? Yes.
Check: must be either `ByteOrder.BIG_ENDIAN` or `ByteOrder.LITTLE_ENDIAN`.
Default: `ByteOrder.LITTLE_ENDIAN`
Keyword: `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: `int`
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: `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: `java.time.Duration`
Optional? Yes.
Check: > zero.
Default: No default.
Keyword: `reconnectTime`

Name: Heartbeat delay.
Description: Send heartbeats at this millisecond rate. If set to zero, heartbeating is not performed.
Type: `java.time.Duration`
Optional? Yes.
Check: ≥ zero.
Default: Zero (no heart-beating).
Keyword: `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: `java.time.Duration`
Optional? Yes.
Check: ≥ zero.
Default: Zero (wait indefinitely).
Keyword: `heartbeatReplyDelay`

Name: Can Pause Flag.
Description: Specifies whether the connection may be paused or not.
Type: `boolean`
Optional? Yes.
Check: None.
Default: false (connection may not be paused).
Keyword: `canPause`

Name: Pause duration.
Description: Pause connection for this maximum time.
Type: `java.time.Duration`
Optional? Yes but must be specified if `canPause` is `true`.
Check: ≥ zero.
Default: No default value if this property is required; zero if not required.
Keyword: `pauseTime`

Name: Message backlog size.
Description: Paused connections will backlog at most this many messages. Once the maximum is reached, messages will be discarded to keep the backlog size at the maximum.
Type: `int`
Optional? Yes but must be specified if `canPause` is `true`.
Check: > zero.
Default: No default value if this property is required; zero if not required.
Keyword: `maxBacklogSize`

Name: Discard policy.
Description: Policy for discarding messages when the message backlog maximum is reached.
Type: `net.sf.eBus.config.EConfigure.DiscardPolicy`
Optional? Yes but must be specified if `canPause` is `true`.
Check: Must be either `DiscardPolicy.OLDEST_FIRST` or `DiscardPolicy.YOUNGEST_FIRST`.
Default: No default value if this property is required; null if not required.
Keyword: `discardPolicy`

Name: Idle time.
Description: Pause when no messages are sent or received for this time.
Type: `java.time.Duration`
Optional? Yes but must be specified if `canPause` is `true`.
Check: ≥ zero.
Default: No default value if this property is required; zero if not required.
Keyword: `idleTime`

Name: Maximum connect time.
Description: Pause after being connected for this time.
Type: `java.time.Duration`
Optional? Yes but must be specified if `canPause` is `true`.
Check: ≥ zero.
Default: No default value if this property is required; zero if not required.
Keyword: `idleTime`

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.jsonConfig.file=<conf file path>
```

Name: Selector threads list.
Description: a JSON array of `SelectorInfo` configurations. The selector names should be unique (no duplicates). All duplicates are logged and ignored. The selector thread name is used to retrieve the remaining keys.
Type: JSON array of `SelectorInfo` configurations.
Optional? Yes.
Check: None.
Default: A single blocking selector thread with `java.lang.Thread.NORM_PRIORITY`.
Keyword: `selectors`

Name: Selector thread name.
Description: Unique selector thread name.
Type: `String`
Optional? No.
Check: Must be a non-null, non-empty string.
Default: No default
Keyword: `name`

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). Matches a `net.sf.eBus.config.ThreadType` text name.
Default: No default
Keyword: `type`

Name: Default selector thread flag.
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 default selector thread).
Keyword: `isDefault`

Name: Thread priority
Description: The selector thread is assigned this Java thread priority value.
Type: `int`
Optional? Yes.
Check: Must be ≥ `java.lang.Thread.MIN_PRIORITY` and ≤ `java.lang.Thread.MAX_PRIORITY`.
Default: `java.lang.Thread.NORM_PRIORITY`
Keyword: `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: `long`
Optional? Yes.
Check: > zero
Default: `net.sf.eBus.net.ENetConfigure.DEFAULT_SPIN_LIMIT`
Keyword: `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: `long`
Optional? Yes.
Check: > zero
Default: `net.sf.eBus.net.ENetConfigure.DEFAULT_PARK_TIME`
Keyword: `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.jsonConfig.file=<conf file path>
# NOTE: selectors must be defined first in the property file.

# Two separate select threads are used: a spinning market data, and a
# spin+park selector for stock orders.
selectors : [

  # Market data selector is low latency, so uses a spinning selector thread.
  {
    name : mdSelector
  type=spinning
  isDefault=false
  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.
{
  name : orderSelector
  type="spin+park"
  isDefault=false
  priority=7
  spinLimit=3000000
  parkTime=1000ns
}
# All other utility channels use a low-priority blocking selector.
{
  name : utilitySelector
  type=blocking
  isDefault=true
  priority=3
}
```

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 v. 5.1.0 supports using **typesafe** JSON configuration using the **HOCON** format. eBus service and client connections may 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.jsonConfig.file=<conf 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 JSON typesafe properties format.

```
// Selectors for connections and services listed below.
selectors : [
    {
        name : mdSelector
        type : spinning
        isDefault : false
        priority : 9
    },
    {
        name : orderSelector
        type : "spin+park"
        isDefault : false
        priority : 7
        spinLimit : 3000000
        parkTime : 1000ns
    },
    {
        name : utilitySelector
        type : blocking
        isDefault : true
        priority : 3
    }
]

connections : [
    {
        name : marketData
        connectionType : UDP
        host : "192.168.3.1"
        port : 10000
        inputBufferSize : 8192
        outputBufferSize : 8192
        byteOrder : BIG_ENDIAN
        messageQueueSize : 0
        selector : mdSelector
        // No heartbeating, no reconnecting.
        heartbeatDelay : 0
        reconnect : false

        // Market data connections may not be paused.
        canPause : false

    },
    {
        name : stockOrders
        // connectionType defaults to TCP.
        host : "StocksRUs.com"
        port : 10001
        bindPort : 55000
        inputBufferSize : 1024
        outputBufferSize : 1024
        byteOrder : BIG_ENDIAN
        messageQueueSize : 10
        selector : orderSelector
        reconnect : true
        reconnectTime : 500ms

        // Order connections may be paused - for demonstration purposes only.
        canPause : true

        // Pause configuration.
        pause : {
            pauseTime : 5m            // Pause for at most 5 minutes or …
            resumeOnBacklogSize : 10 // … 10 messages are queued.
            maxBacklogSize : 20      // Queue up at most 20 messages.
            discardPolicy : YOUNGEST_FIRST
            idleTime : 1m             // Pause after being idle 1 minute or …
            maxConnectTime : 2m      // … connected for 2 minutes.
        }
    }
]

services : [
    {
        name : monitorService
        connectionType : UDP
        port : 8912
        addressFilter : [
            "192.168.3.2", "192.168.3.3:550001"
        ]
        inputBufferSize : 1024
        outputBufferSize : 1024
        byteOrder : BIG_ENDIAN
        messageQueueSize : 10000
        serviceSelector : utilitySelector
        connectionSelector : utilitySelector
        heartbeatDelay : 30s
        heartbeatReply : 500ms

        // Accepted connections may be paused.
        canPause : true

        pause : {
            pauseTime : 10m     // Pause time is at most 10 minutes.
            maxBacklogSize : 25 // Queue up at most 25 messages.
        }
    }
]
```

Step 6: Building Servers and Connections.

eBus servers and connections may be started programmatically using `ServerBuilder` and `ConnectionBuilder` classes. The builder API allows the program to specifically set just those parameters that are required and to leave the rest to their optional settings. Builders are instantiated using a no argument default constructor. The builder default settings are:

Parameter	ServerBuilder	ConnectionBuilder
connection type	`EConfigure.ConnectionType.TCP`	`EConfigure.ConnectionType.TCP`
address filter	`null` - no address filter applied	NA
bind port	NA	`ERemoteApp.ANY_PORT`
input buffer size	`AsyncChannel.DEFAULT_BUFFER_SIZE`	`AsyncChannel.DEFAULT_BUFFER_SIZE`
output buffer size	`AsyncChannel.DEFAULT_BUFFER_SIZE`	`AsyncChannel.DEFAULT_BUFFER_SIZE`
byte order	`ByteOrder.LITTLE_ENDIAN`	`ByteOrder.LITTLE_ENDIAN`
max message queue size	0 - unlimited queue size	0 - unlimited queue size
server selector thread	`AsyncChannel.defaultSelector`	NA
connection selector thread	`AsyncChannel.defaultSelector`	`AsyncChannel.defaultSelector`
heartbeat delay	zero milliseconds: no heartbeating	zero milliseconds: no heartbeating
heartbeat reply delay	zero milliseconds: indefinite wait	zero milliseconds: indefinite wait
reconnect flag	NA	false - no auto-reconnect.
reconnect delay	NA	0 - no auto-reconnect

What follows are two examples showing how `ServerBuilder` and `ConnectionBuilder` may be used to open an `EServer` and `ERemoteApp`:

```java
final AddressFilter filter = ...;
final EConfigure.ServerBuilder builder = EConfigure.serverBuilder();

EServer.openServer(builder.name("AppServer") // Required
                          .connectionType("UDP") // Required, defaults to "TCP"
                          .port(6789)        // Required
                          .addressFilter(filter)
                          .inputBufferSize(1_024)
                          .outputBufferSize(1_024)
                          .byteOrder(ByteOrder.BIG_ENDIAN)
                          .messageQueueSize(100)
                          .serviceSelector("svcSelector")
                          .connectionSelector("connSelector")
                          .heartbeatDelay(60_000L)
                          .heartbeatReplyDelay(30_000L)
                          .build()); // creates EConfigure.Service
```

```
final EConfigure.ConnectionBuilder builder = EConfigure.connectionBuilder();
ERemoteApp.openConnection(builder.name("Conn0")     // Required
                                 .connectionType("UDP") // Required, defaults to "TCP"
                                 .address(address)  // Required
                                 .bindPort(16421)
                                 .inputBufferSize(4_996)
                                 .outputBufferSize(8_192)
                                 .byteOrder(ByteOrder.BIG_ENDIAN)
                                 .messageQueueSize(100)
                                 .selector("connSelector")
                                 .reconnect(true)
                                 .reconnectDelay(500L)
                                 .heartbeatDelay(0L)  // Heartbeating off
                                 .heartbeatReplyDelay(0L)
                                 .build()); // creates EConfigure.RemoteConnection
```

An eBus server and connection are closed by calling:

```
EServer.closeServer(port);
ERemoteApp.closeConnection(address);
```

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

```
EServer.closeAllServers();
ERemoteApp.closeAllConnections();
```

Building Secure Servers and Connections

Secure TCP connections, both server and connection, are built by adding the following two properties:

```
final javax.net.ssl.SSLContext secureContext = ...;
EServer.openServer(builder.name("AppServer")
                          .port(6789)
                          .connectionType(EConfigure.ConnectionType.SECURE_TCP)
                          .sslContext(secureContext)
                          ...

ERemoteApp.openConnection(builder.name("Conn0")
                                 .address(address)
                                 .connectionType(EConfigure.ConnectionType.SECURE_TCP)
                                 .sslContext(secureContext)
                                 ...
```

Please note that it is the eBus user's responsibility to create the `SSLContext` instance in a secure fashion and provide that instance to eBus. eBus uses that context to establish a secure TCP connection based on SSL/TLS.

Step 7: 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.client.IESubscribeFeed;
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.builder())
                .target(this)
                .messageKey(ConnectionMessage.MESSAGE_KEY)
                .scope(FeedScope.LOCAL_ONLY)
                .condition(m -> ((ConnectionMessage) m).serverPort == mPort);
        mConnectFeed.subscribe();
    }

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

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

    @Override public void notify(final ENotificationMessage msg,
                                 final IESubscribeFeed 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));
    }
}
```

Pausing Connections

eBus v. 5.1.0 introduces connection pause. A *client* eBus may request that a connection be paused for a given duration and that the far-end queue up undelivered messages to a maximum backlog size. The far-end server eBus may accept or reject the pause request. If accepted, the far-end response contains the pause duration and backlog size allowed. These values will be ≤ the requested values. When the client receives this acceptance pause response, the connection is closed. The client then reconnects after the agreed upon delay and receives the undelivered messages (if any).

The client may resume the connection earlier than the agreed upon pause delay if it's local message queue reaches a configured limit. For example, both ends agree to pause for 5 minutes but the client eBus is configured to reconnect when the backlog contains 10 application messages (system messages are not used to calculate queue size). When the backlog has 10 application messages after 3 minutes, the client eBus resumes the paused connection.

Client connections are automatically paused after a specified idle time during which no messages are sent or received *or* after a maximum connect time. This is to prevent a busy connection from keeping the connection up permanently.

The pause feature is targeted for mobile devices which cannot maintain a network connection for any length of time without excessive battery drain.

Multicast Connections

eBus release 5.5.0 introduces multicast connections between eBus applications allowing *notification* messages to be shared between those applications joined to the same multicast group. The eBus multicast connection is built around a multi-feed and used to play a single role: either multicast publisher or multicast subscriber.

A multicast publisher uses an `EMulti`*`Subscribe`*`Feed` to receive remotely scoped[10] notifications from eBus and then posts the messages to the multicast group. Conversely a multicast subscriber receives messages from the multicast group and publishes to eBus using an `EMulti`*`Publish`*`Feed`. The notification messages allowed across a multicast connection are defined by its multi-feed(s). These feeds are set when the multicast connection is created. The following sections show how to define a multicast connection using a typesafe JSON configuration (in HOCON format) and then using the eBus API to create the same connection.

Multicast JSON Configuration[11]

The easiest way to define and open multicast connections is by placing them into typesafe properties file and then loading that file on application start:

```
  -Dnet.sf.eBus.config.jsonFile=<conf file path>
```

The format is similar to that used in the above section JSON Configuration. This example contains two multicast connections in the application: one to publish `TextMessage`s and the other to receive `Trade`s. The `selectors` property is used for both direct and multicast connections and so is not shown in this example. Assume that this property is set and defines `mcastSelector`. Required properties are highlighted in **green**.

[10] A publish feed scoped as `Feed.FeedScope.LOCAL_AND_REMOTE` or `REMOTE_ONLY`.

[11] eBus multicast connection can only be defined using typesafe JSON configuration.

```
multicastGroups : [
  # Publisher multicast connection.
  {
    name : MCAST-PUB
    mcastRole : PUBLISHER
    multicastGroup : "230.0.0.0"
    targetPort : 5000
    networkIF : en8
    protocolFamily : INET
    bindPort : 5001
    order : LITTLE_ENDIAN
    selector : "mcastSelector"
    inputSize : 512
    outputSize : 512

    # EMultiPublishFeeds define notification messages posted to multicast
    # group.
    multicastKeys : [
      multifeedType : LIST
      messageClass : com.acme.personal.TextMessage
      subjectList : [
        "Tom", "Dick", "Harry"
      ]
    ]
  },
  # Subscriber multicast connection.
  {
    name : MCAST-SUB
    mcastRole : SUBSCRIBER
    multicastGroup : "230.0.0.0"
    targetPort : 5011
    networkIF : en8
    protocolFamily : INET
    bindPort : 5010
    order : LITTLE_ENDIAN
    selector : "mcastSelector"
    inputSize : 512
    outputSize : 512

    # EMultiSubscribeFeeds define notification messages received from
    # multicast group. In this example receive trade notifications for
    # stock symbols A-M, inclusive.
    multicastKeys : [
      multifeedType : QUERY
      messageClass : com.acme.marketData.Trade
      subjectQuery : "[A-M][A-Z]*"
      # Allow newly defined symbols to be automatically added to the feed.
      isDynamic : true
    ]
  }
]
```

Building eBus Multicast Connections

The same multicast connection is now defined dynamically using the eBus multicast connection API. Again required settings are in **green**.

```
import com.acme.marketData.Trade;
import com.acme.personal.TextMessage;
import com.google.common.collect.ImmutableList;
import net.sf.eBus.config.EConfigure;

final EConfigure.MulticastBuilder pubBuilder =;
final EConfigure.MulticastBuilder subBuilder = EConfigure.multicastBuilder();
final InetAddress group = InetAddress.getByName("230.0.0.0");
final NetworkInterface netIF = NetworkInterface.getByName("en8");
final String textClass = TextMessage.getCanonicalName();
final List<String> textSubjects = ImmutableList<>.of("Tom", "Dick", "Harry");
final String tradeClass = Trade.getCanonicalName();
final String tradeQuery = "[A-M][A-Z]*";
final EConfigure.McastNotifyConfig listFeed =
    (EConfigure.notificationBuilder()).feedType(EConfigure.MultifeedType.LIST)
                                      .messageClass(textClass)
                                      .subjectList(textSubjects)
                                      .build();
final EConfigure.McastNotifyConfig queryFeed =
    (EConfigure.notificationBuilder()).feedType(EConfigure.MultifeedType.QUERY)
                                      .messageClass(tradeClass)
                                      .subjectList(tradeQery)
                                      .isDynamic(true)
                                      .build();
final List<EConfigure.McastNotifyConfig> textKeys =
    ImmutableList<>.of(listFeed);
final List<EConfigure.McastNotifyConfig> tradeKeys =
    ImmutableList<>.of(queryFeed);
final EConfigure.MulticastConnection pubConfig =
    (EConfigure.multicastBuilder()).name("MCAST-PUB")
                                   .role(EConfigure.MulticastRole.PUBLISHER)
                                   .group(group)
                                   .targetPort(5000)
                                   .networkInterface(netIF)
                                   .protocolFamily(StandardProtocolFamily.INET)
                                   .bindPort(5001)
                                   .byteOrder(ByteOrder.LITTLE_ENDIAN)
                                   .inputBufferSize(512)
                                   .outputBufferSize(512)
                                   .notifications(textKeys)
                                   .build();
final EConfigure.MulticastConnection pubConfig =
    (EConfigure.multicastBuilder()).name("MCAST-SUB")
                                   .role(EConfigure.MulticastRole.SUBSCRIBER)
                                   .group(group)
                                   .targetPort(5011)
                                   .networkInterface(netIF)
                                   .protocolFamily(StandardProtocolFamily.INET)
                                   .bindPort(5010)
                                   .byteOrder(ByteOrder.LITTLE_ENDIAN)
                                   .inputBufferSize(512)
                                   .outputBufferSize(512)
                                   .notifications(tradeKeys)
                                   .build();

// Open multicast connection based on given configuration.
EMulticastConnection.openConnection(pubConfig);
EMulticastConnection.openConnection(subConfig);
```

Monitoring eBus Connections

If there is a need to dynamically monitor eBus connection status (client, server, or multicast) then subscribe to the following message keys, one for each eBus connection type:

- **client**: `net.sf.eBus.client.ConnectionMessage.MESSAGE_KEY`
- **server**: `net.sf.eBus.client.ServerMessage.MESSAGE_KEY`
- **multicast**: `net.sf.eBus.client.MulticastMessage.MESSAGE_KEY`

Subscribe to the messages as follows (*note* - these messages are published to the local eBus JVM only):

```
(ESubscribeFeed.builder()).target(subscriber)
                          .messageKey(ConnectionMessage.MESSAGE_KEY)
                          .scope(FeedScope.LOCAL_ONLY)
                          .condition(condition)
                          .build();
```

eBus sends updates as a connection goes through the process of connecting, disconnecting, accepting, or joining a multicast group. Since there is no way to retrieve the current status (at this time) it is best to put the subscription(s) in place before opening connections or services.

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>` or `-Dnet.sf.eBus.config.jsonFile=<config 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.

Active and Hybrid Objects

The reason why hybrid object feeds must target their parent active object should now be apparent: because the hybrid object shares its parent active object's Dispatcher. While the hybrid object's method processes the message, that message is posted to the active object's queue. So the active object remains single threaded but subordinates message processing to its hybrid objects.

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.[12]

   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

[12] See Appendix E: Configuing Thread Affinity to learn how to "pin" a spinning Dispatcher thread to a core isolated from the operating system.

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 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);
    }
}
```

Special Dispatchers

There are two special, pre-defined Dispatchers which go by the names "swing" and "javafx" (case insensitive). These Dispatchers use the Swing and JavaFX GUI threads, respectively, to deliver eBus messages to those clients associated with the Dispatcher. This means that the client callback is free to update the GUI because the callback code is running on the GUI thread.

These special Dispatchers only support two properties: `isDefault` and `classes`. All other properties are quietly ignored by eBus. The reason is that the underlying GUI threads are implemented and configured by the GUI package and cannot be altered by eBus.

It may be useful to define the GUI thread Dispatcher as the default Dispatcher and then create a separate eBus Dispatcher for non-GUI classes. This way, an application class which updating the display will be assigned to the GUI thread without needing to add that class to the GUI Dispatcher classes property.

Monitoring Dispatchers

eBus release 6.1.0 introduced access to dispatcher run-time statistics by adding the static method `EClient.runTimeStats()` which returns `EClient.RunTimeStats` list. These stats apply to the existing `EObject` instances and include: minimum time on Dispatcher thread (in nanoseconds), maximum time, total time, number of times on Dispatcher thread, average time, Dispatcher maximum run quantum, and number of times eBus object exceeded Dispatcher run quantum.

eBus release 6.2.0 expanded run queue thread and eBus object monitoring with the singleton class `net.sf.eBus.client.monitor.RQMonitor`. This monitoring includes:

- Reporting run queue thread statistics at a configurable interval. These reports are `net.sf.eBus.client.monitor.RQThreadReport` on `RQMonitor.REPORT_SUBJECT`.
- Reporting eBus object statistics at the same configurable interval as the run queue thread report. These reports are `net.sf.eBus.client.monitor.EBusObjectReport` notifications on `RQMonitor.REPORT_SUBJECT`. Note that this report contains the same information as `EClient.runTimeStats()` method.
- Reporting when an eBus object is overrunning the dispatcher maximum quantum (plus a configurable limit). An alarm is raised via `net.sf.eBus.client.monitor.ThreadOverrunUpdate` notification on `RQMonitor.RQALARM_SUBJECT`.
- Reporting when a run-ready eBus object is denied access to a run queue thread for too long (again how long is configurable). An alarm is raised via `net.sf.eBus.client.monitor.ThreadDenialUpdate` notification on `RQMonitor.RQALARM_SUBJECT`.

Alarms are raised when a condition transitions from cleared to alarmed or from alarmed to cleared. The run queue report contains the current thread alarm state.

`RQMonitor` also implements `net.sf.eBus.util.logging.StatusReporter` interface and will add run queue thread and eBus object stats to the `net.sf.eBus.util.logging.StatusReport` *if* the `RQMonitor` is started and registered with `StatusReport`.

Dispatcher Configuration

Name: Dispatchers list.
Description: a JSON array of `EConfigure.Dispatcher` configurations. Dispatcher names should be unique (no duplicates). All duplicates are logged and ignored. The dispatcher name is used to retrieve the remaining keys. If using Swing or JavaFX GUI Dispatcher, then place the name “swing” or “javafx” here.
Type: `String`
Optional? Yes.
Check: None.
Default: A single blocking dispatcher with `java.lang.Thread.NORM_PRIORITY` and no run quantum. Used for all eBus clients.
Keyword: `dispatchers`

Name: Dispatcher thread name.
Description: Dispatcher thread name unique within the
Type: `int`
Optional? No.
Check: > zero.
Default: `net.sf.eBus.client.eConfigure.DEFAULT_NUMBER_THREADS`
Keyword: `name`

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

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

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

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

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

Name: Client run quantum
Description: Client task processing time quantum.
Type: `java.time.Duration`
Optional? Yes.
Check: > zero.
Default: `net.sf.eBus.client.eConfigure.DEFAULT_QUANTUM`
Keyword: `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: `isDefault`

Name: Dispatcher client classes.
Description: Clients instantiated from the specified classes are assigned to this dispatcher.
Type: JSON typesafe array 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: `classes`

The following shows how to configure eBus Dispatchers using JSON:

```
dispatchers : [
    {
        name : mdDispatcher
        numberThreads : 1
        runQueueType : spinning
        priority : 9
        quantum : 10000ns
        isDefault : false
        classes : [ "com.acme.trading.MDHandler" ]
        threadAffinity {        // optional, selector thread core affinity
            affinityType : CPU_ID // required, core selection type.
            cpuId : 7             // required for CPU_ID affinity type
            bind : true           // optional, defaults to false
            wholeCore : true      // optional, defaults to false
    },
    {
        name : defaultDispatcher
        numberThreads : 1
        runQueueType : "spin+park"
        priority : 4
        quantum : 100000ns
        isDefault : true
        spinLimit : 2500000
        parkTime : 1000ns
    }
]
```

eBus release 5.8.0 introduced thread affinity for dispatcher threads. This affinity is created when adding `threadAffinity` property to the dispatcher configuration.

Thread affinity is recommended especially for dispatcher threads configured as spinning since a thread doing hard spinning is not amenable to pre-emption. Also be sure to isolate the selected core from operating system use. Otherwise the spinning thread may still be pre-empted by the OS for its own particular use.

Please note: threadAffinity property is only supported in JSON configuration files and *not* in Java properties files.

See Appendix E: Configuring Thread Affinity for a detailed discussion on using thread affinity.

eBus release 6.4.1 added method `net.sf.eBus.client.EFeed.createDispatcher(Configure.Dispatcher)` allowing a Dispatcher and its subordinate thread(s) to be created dynamically post-JVM initialization. The Dispatcher configuration is created using a `DispatcherBuilder` instance which can be acquired from `net.sf.eBus.config.EConfiure.dispatcherBuilder()` static method. The Dispatcher configuration is that same a noted in section Dispatcher Configuration.

Programmatic Dispatcher Configuration

An eBus dispatcher can be created using `net.sf.eBus.config.EConfigure.DispatcherBuilder`. The following example shows how this can be done creating a single-thread `Dispatcher` pinned to a specific CPU core and spinning on the ready `EClient` queue. This dispatcher is used for a singleton `MarketDataHandler` instance.

NOTE: This example only works ***if*** the dispatcher is created *before* `MarketDataHandler` is instantiated. If not, the `MarketDataHandler` instance will be assigned to the default `Dispatcher`.

```
import net.sf.eBus.client.EFeed;
import net.sf.eBus.config.EConfigure.Dispatcher;
import net.sf.eBus.config.EConfigure.DispatcherBuilder;
import net.sf.eBus.config.EConfigure.DispatcherType;
import net.sf.eBus.config.EConfigure.ThreadAffinityConfigure;
import net.sf.eBus.config.ThreadType;

public static void main(final String[] args) {
    final Class[] eclients = new Class[] { MarketDataHandler.class };
    final ThreadAffinityConfigure[] threadAffinity =
        new ThreadAffinityConfigure[] {…};
    final DispatcherBuilder builder = EConfigure.dispatcherBuilder();
    final Dispatcher dispatcher = builder.name("MyDispatcher")
                                          .dispatcherType(DispatcherType.EBUS)
                                          .threadType(ThreadType.SPINNING)
                                          .numberThreads(1)
                                          .isDefault(false)
                                          .classes(eclients)
                                          .threadAffinity(threadAffinity)
                                          .build();

    EFeed.createDispatcher(dispatcher);
}
```

See eBus javadocs for `EConfigure.Dispatcher` and `EConfigure.DispatcherBuilder` for more information on programmatically creating a 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.builder()).target(this)
                                             .messageKey(mSubKey)
                                             .scope(FeedScope.LOCAL)
                                             .build();
        mSubFeed.subscribe();

        // Race condition between feedStatus callback and the next two lines.
        mPubFeed = (EPublishFeed.builder()).target(this)
                                           .messageKey(mPubKey)
                                           .scope(FeedScope.LOCAL)
                                           .build();
        mPubFeed.advertise();
    }

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

    @Override public void feedStatus(final EFeedState feedState,
                                     final IESubscribeFeed 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 IESubscribeFeed 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[13] and then call `EFeed.startupAll()`. This method starts up all registered application objects that were not previously started or implicitly registered.[14] 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.

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

[14] 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.IESubscribeFeed
%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, IESubscribeFeed)
    feedState(feedState: final EFeedState, feed: final IESubscribeFeed)
      [feedState == EFeedState.UP]
        Collecting {}

    // Wait here for the feed to come up. nil is an internal loopback transition.
    feedState(feedState: final EFeedState, feed: final IESubscribeFeed)
        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 IESubscribeFeed)
      [feedState == EFeedState.UP]
        Collecting {}

    // Wait here for the feed to come up.
    feedState(feedState: final EFeedState, feed: final IESubscribeFeed)
        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, IESubscribeFeed)
    data(msg : final ENotificationMessage, feed: final IESubscribeFeed)
      [ctxt.addData(((DataPoint) msg).data)]
        Publishing {}

    // Continue collecting.
    data(msg : final ENotificationMessage, feed: final IESubscribeFeed)
        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 IESubscribeFeed)
        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 IESubscribeFeed)
      [feedState == EFeedState.DOWN]
        FeedDown {}

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

    // Ignore unexpected data points.
    data(msg : final ENotificationMessage, feed: final IESubscribeFeed)
        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() {
    // 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 = (ESubscribeFeed.builder()).target(this)
                                         .messageKey(mSubKey),
                                         .scope(EFeed.FeedScope.LOCAL_ONLY)
                                         .statusCallback(mFsm::feedState)
                                         .notifyCallback(mFsm::data)
                                         .build();
    mSubFeed.subscribe();

    mPubFeed = (EPublishFeed.builder()).target(this)
                                       .messageKey(mPubKey)
                                       .scope(EFeed.FeedScope.LOCAL_ONLY)
                                       .build();
    mPubFeed.advertise();
}

// Ignore publish status updates.
@Override public void publishStatus(final EFeedState feedState,
                                    final IEPublishFeed 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.

Going Mobile[15]

eBus release 6.6.0 provides changes allowing the eBus API to be used on an Android mobile device. These changes include:

- Remove use of `java.lang.management` package since this is not supported on Android Runtime (ART).
- Set unique JVM identifier to `UUID.randomUUID()` rather than `ManagementFactory.getRuntimeMXBean().getName()`.
- Add `util` package method `ERuntime.isAndroid()` to determine if eBus is running on ART rather than a Java virtual machine.
- Remove dependence on `javassist` API since that generates Java byte code which ART uses Dalvik byte code (done in eBus release 6.5.0 with `net.sf.eBus.messages.type.InvokeMessageCompiler` and `InvokeMessageType`). If eBus detects it is running on ART (see previous point) it automatically sets the message compiler to `InvokeMessageCompiler`.

eBus release 6.6.0 has been successfully tested on an Android emulator and demonstrated the ability to establish a clear text TCP and UDP connection to a JVM-based eBus application, both sending and receiving messages between the eBus applications. What was *not* tested was the ability to establish a secure TCP (using TLS), or secure UDP (using DTLS) connection.

The eBus API is used on Android just as on a JVM. The main difference is that `-Dnet.sf.eBus.config.jsonFile=`*eBus config file* cannot be used on Android. Dispatcher and Selector threads must be configured programmatically on Android. See sections Dynamic Selector Definition and Programmatic Dispatcher Configuration for more information.

Since eBus delivers inbound messages and events on a Dispatcher thread, it is up to the Android developer to use `android.os.Handler`, `android.os.Looper`, and `android.os.Message` classes to post updates to the Android user interface (UI) thread.

[15] From "Going Mobile" lyrics by Peter Townsend.

Time, Gentlemen!

eBus dispatcher delivers messages to application objects such that an object is accessed by only one *eBus thread* at a time so that the application object is effectively single threaded. But if the application object interacts with other non-eBus threads, that single threaded guarantee is lost.

One common use of a non-eBus thread is either `java.util.Timer` or `java.util.concurrent.ScheduledExecutorService` (preferred). There is a definite need for timers in an application. When the timer expires and the target method is called, the application has two choices: either synchronize data access or use `EClient.dispatch(Runnable, EClient)` to transfer control from the non-eBus thread back to an eBus thread. But doing that introduces delay.

eBus release 7.3.0 introduces `net.sf.eBus.time.EScheduledExecutor`. This class has a similar interface to `java.util.concurrent.ScheduledExecutorService` but requires an `EObject` as the second argument and does not support scheduling a `java.util.concurrent.Callable` nor implements `java.util.Executor` or `java.util.ExecutorService` interfaces. When `EScheduledExecutor` detects an active timer expiration, it passes the scheduled task and application object to `EClient.dispatch`.[16] This means the timer task is processed by an eBus thread.

There are three `EScheduledExecutor` schedule methods:

- `IETimer schedule(Runnable task, EObject eobject, Duration delay)`
  Submits a single-shot task which expires after the given delay. Use returned IETimer.close() to cancel active timer.
- `IETimer scheduleAtFixedRate(Runnable task, EObject eobject, Duration initialDelay, Duration period)`
  Submits a periodic task which expires for the first time after the initial delay and then repeatedly at the periodic rate. In other words, expirations are `initialDelay`, then `initialDelay + period`, then `initialDelay + (2 * period)`, and so on.
  The given task will continue to be indefinitely executed until one of the following occurs:
  - The task is explicitly canceled via the returned `IETimer` instance.
  - The client is garbage collected.
  - The executor is terminated resulting in all scheduled tasks being canceled.
  - The tasks's execution results in a thrown exception.

  Once a timer is canceled, subsequent executions are suppressed and `isDone` returns `true`.
  If any task execution takes longer than its period, then subsequent executions may start late but will not result in multiple scheduled expirations.
- `IETimer scheduleWithFixedDelay(Runnable task, EObject eobject, Duration initialDelay, Duration delay)`
  Submits a periodic task which expires for the first time after the initial delay and then repeatedly with the given delay between the termination of the previous expiration and the commencement of the next. This means that task execution time does not impact scheduling the subsequent expirations. When the task completes, the next expiration is current time plus delay.

EScheduledExecutor does *not* support a delay or period ≤ zero. A repeating fixed delay or repeating fixed period must be > zero. A zero single shot delay or initial delay must be ≥ zero.

[16] Actually the scheduled task is wrapped within a `EScheduledExecutor.TimerTask` which is used to catch any `Exception` thrown by the scheduled task and automatically cancel the timer when caught.

Creating an EScheduledExecutor

There are two ways to create an `EScheduledExecutor`: programmatically or in the eBus JSON configuration file. Both can be used within the application. Either way every eBus scheduled executor ***must*** have a unique thread name.

Note: eBus provides an executor named "CoreExecutor" in `net.sf.eBus.client.EClient.sCoreExecutor`. This scheduler is a low priority thread and blocks on the next timer expiration providing lower precision timing useful for most application needs.

Property	Type	Required?	Default	Description
name	`String`	Yes		*Unique* thread name.
thread type	`ThreadType`	Yes		Thread type is blocking, spinning, spin+park, or spin+yield
priority	`int`	No	`EConfigure.DEFAULT_PRIORITY`	Thread priority.
spin limit	`long`	Yes if thread type is spin+park or spin+yield		How many times thread will spin waiting for timer expiration before parking or yield.
park time	`long`	Yes if thread type is spin+park.		How long thread will park before spinning again.
thread affinity	`List<ThreadAffinity>` `ThreadAffinity[]`		No thread affinity.	Used to create affinity between thread and processor core(s).

EScheduledExecutor Configuration Properties

Programmatic eBus Scheduler Creation

```
import java.time.Duration;
import net.sf.eBus.config.EConfigure;
import net.sf.eBus.config.EConfigure.ScheduledExecutor;
import net.sf.eBus.config.EConfigure.ScheduledExecutorBuilder;
import net.sf.eBus.time.EScheduledExecutor;

// First build the schedule executor configuration.
final ScheduledExecutorBuilder builder = EConfigure.scheduledExecutorBuilder();
final ScheduledExecutor config = builder.name("FastTimer")
                                        .threadType(ThreadType.SPINPARK)
                                        .priority(8)
                                        .spinLimit(2_500_000)
                                        .parkTime(Duration.ofNanos(500L)
                                        .build();

// Then use that configuration to create the eBus scheduled executor.
final EScheduledExecutor executor = EScheduledExecutor.newScheduledExecutor(config);
```

JSON eBus Scheduler Creation

```
scheduledExecutors : [
  {
    name : "blocking-timer"
    threadType : "blocking"
    priority : 3
  },
  {
    name : "high-priority-timer"
    threadType : "spinning"
    priority : 10
    threadAffinity : [17
      {
        affinityType : CPU_ID
        cpuId = 7
        bind : true
        wholeCore : true
      }
    ]
  },
  {
    name : "low-priority-timer"
    threadType : "spin+park"
    priority : 7
    spinLimit : 2500000
    parkTime : 500 ns
  }
]
```

17 See Appendix E: Configuring Thread Affinity for detailed explanation.

Keeping an Eye on Things

eBus provides monitoring at two levels: application and Dispatcher. Application level allows developer to instrument eBus EObject code which reports both on-going and transient events occurring in the code. Dispatcher level reports run queue thread events and periodically reports (at an application defined rate) the run queue thread performance statistics.

Application Monitoring

Package net.sf.eBusx.monitor provides applications the ability to instrument and track eBus EObject status. Monitor is the central class in this package. The very first step in using this package is to open a Monitor instance with the application "builder plate" and optional attributes:

```
import net.sf.eBusx.monitor.Monitor;

final AppAttributes attributes =(AppAttributes.builder()).set attributes here.build();

// Note: if Monitor instance is already open, then does nothing.
final Monitor monitor = Monitor.openMonitor("MyOwnApp UAT 1",
                                            "MyOwnApp",
                                            "1.2.3",
                                            "Copyright (C) 2024 All Rights Reserved",
                                            "My very own little Java application",
                                            attributes);
```

where AppAttributes is an EField subclass containing attributes specific to this application. The copyright, description, and attributes fields are optional and may be null. This application information is both stored away for later retrieval and published to existing application info subscribers.

The first argument is the application's host name. Please note that this name is *not necessarily the network host name*. Often times a network host name is a obtuse name meaning to the network support staff. Monitor allows a name meaningful to application support to be used.

Please note: once the singleton Monitor instance is opened, it cannot be closed. It remains open as long as the JVM is running.

Instrumenting EObjects

This section describes how to use Monitor to track and report an eBus EObject's on-going and transient status. On-going status is just that: a status that is considered in effect from the moment it is reported to when it is either updated or the eBus object is de-registered from the monitor. A transient status is an event which has no impact on the on-going status.

The first step in instrumenting an eBus object is to register it with Monitor. The monitor instance is acquired either from Monitor.openMonitor (as shown above) or via Monitor.getMonitor. Note that getMonitor returns null if Monitor singleton was not opened. An eBus object is registered as follows (consider this class to implement an EObject interface):

```
(Monitor.getMonitor().register(this);
```

Note: Monitor does *not* maintain a strong reference to this but a weak reference. This means that registered eBus objects can be finalized while still registered to Monitor. Monitor detects this finalization and automatically de-registers the object.

A newly registered eBus object is given the following initial status:

- Action level: `ActionLevel.NO_ACTION`
- Action name: `"Registered"`
- Action message: `"Registered with monitor subsystem"`

A registered eBus object is assigned a unique `MonitorId` which is used for reporting purposes only. More about this identifier below.

Once registered, an eBus object may update its on-going (persistent) state as follows:

```
final Monitor monitor = Monitor.getMonitor();

monitor.update(ActionLevel.ACTION_REQUIRED,
               "Market data feed",
               "Ticker plant market data feed is DOWN",
               this);
```

When this condition is resolved, the eBus object should report this fact:

```
monitor.update(ActionLevel.NO_ACTION,
               "Market data feed",
               "Ticker plant market data feed is up",
               this);
```

An example of a transient status update would be posted when the market data feed is down, reporting how long the condition has been in effect:

```
monitor.transientStatus(ActionLevel.ACTION_REQUIRED,
                        "Market data feed",
                        "Ticker plant market data feed down for " + downTimeInterval,
                        this);
```

If a registered eBus object is to be discarded prior to application termination, it should be de-registered from `Monitor`:

```
monitor.deregister(this);
```

A de-registered eBus object is given the following final status:

- Action level: `ActionLevel.NO_ACTION`
- Action name: `"Deregistered"`
- Action message: `"Deregistered from monitor subsystem"`

`Monitor` reports object registration, on-going updates, transient updates, and de-registration using `MonitorUpdate` notification message which contains the following fields:

- host name (`String`): set in `openMonitor.`
- application name (String): set in `openMonitor.`
- monitor instance (`MonitorID`): assigned to eBus object when registered.
- update type (`UpdateType`): specifies if eBus object is registered, on-going status update, transient update, or de-registered.
- action level (`ActionLevel`): set in `update` or `transientStatus` call.

- action name (`String`): set in `update` or `transientStatus` call.
- action message (`String`): set in `update` or `transientStatus` call.

These MonitorUpdate messages are published to the subject "/eBus/monitor/*host name*/*app name*" where *host name* and *app name* are those used to open the `Monitor` instance.

`Monitor` also replies to `ApplicationInfoRequest` and `MonitoredObjectRequest` messages. `ApplicationInfoRequest` reply (`ApplicationInfoReply`) contains the application host name, application name, etc. `MonitoredObjectRequest` reply (`MonitoredObjectReply`) contains the latest `PersistentStatusMessage` for all currently registered eBus objects.

This leads to the next section.

Monitoring eBus applications

The first step is to **establish a remote connection** to all the eBus applications which will be monitored. Of course the local JVM may be monitored as well.

The second step is to subscribe to those `Monitor`-published notifications in which you are interested. As mentioned above, `Monitor` publishes updates using a subject based on the configured host name and application name. If `ESubscribeFeed` is used to subscribe to `Monitor` updates, then a subscription must be created for all host name, application name pairs. This means that monitored hosts and applications are fixed and known prior to subscribing. If this list is dynamic, there must be a way to add or remove pairs.

eBus provides a way to subscribe to all monitor update subjects rather that to each specific host, application name pair: `EMultiSubscribeFeed` using an eBus pattern as the "subject". The following code demonstrates how to use an `EMultiSubscribeFeed` to receive monitor updates:

```
import static net.sf.eBus.client.EFeed.FeedScope;
import net.sf.eBus.client.EMultiSubscribeFeed;
import net.sf.eBus.client.IESubscribeFeed;
import net.sf.eBus.util.regex.Pattern;
import net.sf.eBusx.monitor.ApplicationInfo;
import net.sf.eBusx.monitor.Monitor;
import net.sf.eBusx.monitor.MonitorUpdate;

private EMultiSubscribeFeed mAppInfoFeed;
private EMultiSubscribeFeed mUpdateFeed;

@Override public void startup() {
    final Pattern multiUpdateSubject =
        // Put ".+" in host and application name subject portions to accept updates
        // from all hosts and applications.
        Pattern.compile(String.format(Monitor.MONITOR_UPDATE_FORMAT, ".+", ".+");

    mAppInfoFeed =
        (EMultiSubscribeFeed.build()).target(this)
                                     .messageClass(ApplicationInfo.class)
                                     .scope(FeedScope.LOCAL_AND_REMOTE)
                                     .query(multiUpdateSubject)
                                     .statusCallback(this::onAppInfoFeedStatus)
                                     .notifyCallback(this::onAppInfoUpdate)
                                     .build();
    mUpdateFeed =
        (EMultiSubscribeFeed.build()).target(this)
                                     .messageClass(MonitorUpdate.class)
                                     .scope(FeedScope.LOCAL_AND_REMOTE)
                                     .query(multiUpdateSubject)
                                     .statusCallback(this::onUpdateFeedStatus)
                                     .notifyCallback(this::onMonitorUpdate)
                                     .build();
}

private void onAppInfoFeedStatus(final EFeedState state, final IESubscribeFeed feed) {
    …
}

private void onAppInfoUpdate(final ApplicationInfo info, final IESubscribeFeed feed) {
    …
}

private void onUpdateFeedStatus(final EFeedState state, final IESubscribeFeed feed) {
    …
}

private void onMonitorUpdate(final MonitorUpdate info, final IESubscribeFeed feed) {
    …
}
```

It is recommended that subscriptions be put into place *before* requesting latest monitor updates which is the next step.

The third step is requesting the latest `Monitor` updates. Unlike monitor notification subject, the monitor request subject are fixed: `Monitor.APP_INFO_REQUEST_SUBJECT` and `Monitor.ONGOING_REQUEST_SUBJECT`. It is recommended that these request subjects be put into place on start-up but the requests themselves not be made until the repliers are known to be up. The following code demonstrates how to do this:

```
import static net.sf.eBus.client.eFeed.FeedScope;
import net.sf.eBus.client.ERequestFeed;
import net.sf.eBus.client.IERequestFeed;
import net.sf.eBus.messages.EMessageKey;
import net.sf.eBusx.monitor.ApplicationInfoRequest;
import net.sf.eBusx.monitor.Monitor;
import net.sf.eBusx.monitor.MonitorObjectRequest;

private ERequestFeed mAppInfoRequestFeed;
private ERequestFeed.ERequest mAppInfoRequest;
private ERequestFeed mMonitorRequestFeed;
private ERequestFeed.ERequest mMonitorRequest;

@Override public void startup() {
    final EMessageKey appInfoRequestKey =
        new EMessageKey(
            ApplicationInfoRequest.class, Monitor.APP_INFO_REQUEST_SUBJECT);
    final EMessageKey ogRequestKey =
        new EMessageKey(
            MonitoredObjectRequest.class, Monitor.ONGOING_REQUEST_SUBJECT);

    mAppInfoRquestFeed =
        (ERequestFeed.builder()).target(this)
                                .messageKey(appInfoRequestKey)
                                .scope(FeedScope.LOCAL_AND_REMOTE)
                                .statusCallback(this::onAppInfoRequestFeedStatus)
                                .replyCallback(this::onAppInfoReply)
                                .build();
    mAppInfoRequestFeed.subscribe();

    mMonitorRequestFeed =
        (ERequestFeed.builder()).target(this)
                                .messageKey(ogRequestKey)
                                .scope(FeedScope.LOCAL_AND_REMOTE)
                                .statusCallback(this::onMonitorFeedStatus)
                                .replyCallback(this::onMonitorStatusReply)
                                .build();
    mMonitorRequestFeed.subscribe();
}

private void onAppInfoRequestFeedStatus(final EFeedState state,
                                        final IERequestFeed feed) {
    if (state == EFeedState.UP) {
        final ApplicationInfoRequest.Builder builder =
            ApplicationInfoRequest.builder();

        mAppInfoRequest =
            mAppInfoRequestFeed.request(
                builder.subject(Monitor.APP_INFO_REQUEST_SUBJECT).build());
    }
}

private void onAppInfoReply(final int remaining,
                            final EReplyMessage msg,
                            final ERequestFeed.Request request) {
    // Process application information reply.
}

private void onMonitorFeedStatus(final EFeedState state,
                                 final IERequestFeed feed) {
    if (state == EFeedState.UP) {
        final MonitoredObjectRequest.Builder builder =
            MonitoredObjectRequest.builder();

        mMonitorRequest =
            mMonitorRequestFeed.request(
                builder.subject(Monitor.ON_GOING_REQUEST_SUBJECT).build());
    }
}

private void onMonitorStatusReply(final int remaining,
                                  final EReplyMessage msg,
                                  final ERequestFeed.Request feed) {
    // Process monitored object status reply.
}
```

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 the class `public final` fields. 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 sequence number. Added only for reliable UDP protocol.
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

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

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

`boolean/Boolean`	1 bit	Booleans are stored as a single bit within a 64-bit, signed integer. This `long` value is only serialized if the message/field class has at least one boolean field.

`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.

`java.io.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()`

`java.lang.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.util.UUID`	16	`Pos Sz Description`
		`  0  7 UUID most-significant bits.`
		`  8 15 UUID least-significant bits.`
		De-serialized using
		`new UUID(long, long)`

`java.time.Duration`	8	Serialized using `Duration.toNanos().` De-serialized using `Duration.ofNanos().`

`java.time.Instant`	16	Serialized using `Instant.getEpochSecond(),`
		`Instant.getNano() (as long)`
		De-serialized using
		`Instant.ofEpochSecond(long, long).`

Type	Length	Description
`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()`.
`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)`.

`org.decimal4j.api.Decimal`	12	`Pos Sz Description`
		`  0  8 Decimal.unscaledValue()`
		`  8  4 Decimal.getScale()`
		De-serialized using `DecimalFactory.valueOfUnscaled(long, int)`.

Type	Length	Description
`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``[]` 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 `IEPublishFeed` or `IEReplyFeed` 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 `IESubscribeFeed`; 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: eBus Network Programming

The `net.sf.eBus.net` package adds an asynchronous layer between `java.nio` and an application. Understanding eBus async network programming requires first understanding Java NIO. Java NIO is centered on three classes: `SelectableChannel`, `SelectionKey`, and `Selector`.

Three `SelectableChannel` subclasses: `SocketChannel`, `ServerSocketChannel`, and `DatagramChannel` are used to read and write bytes or accept new connections. But when is a channel ready to perform an I/O operation?

That is where `Selector` comes in. `Selector` watches one or more registered `SelectableChannel`s and determines which operations are ready to be performed on which channel. But how is a channel registered with `Selector`?

`SelectionKey` is used to connect a channel with a selector. A `SelectionKey` instance is created when the selectable channel is registered (via `SelectableChannel.register(selector, opsMask)` method) with the `Selector` for the given initial operations bit mask. Once registered the operations bit mask may be modified directly using `SelectionKey.interestOps(int)`.

Java NIO allows a single selector to watch multiple channels for ready I/O operations. The application calls `Selector.select()` which returns the number of I/O ready channels. Then `Selector.selectedKeys()` which returns the I/O ready `SelectionKey` set. For each I/O ready key `SelectionKey.readyOps()` is called to retrieve the ready I/O operations mask. And that mask tells the application which I/O operation to perform on the channel.

eBus async network API wraps these Java NIO components into its own three classes and so simplifying the interface :

1. `AsyncChannel`: encapsulates a `SelectableChannel`.
2. `SelectorThread`: encapsulates a `Selector`.
3. Listener interface: application class implements the appropriate interface in order to receive callbacks from the `AsyncChannel`. There is a different interface for TCP socket, TCP server socket, and UDP socket.

eBus async API implements the Java NIO API as follows:

1. `SelectorThread` watches registered socket channels for ready I/O operations.
2. When `SelectorThread` detects ready I/O operations, it calls each `SelectionKey`'s affiliated `AsyncChannel`. An `AsyncChannel` affiliation with a `SelectionKey` is accomplished using `SelectionKey.attach(Object)`.
3. `AsyncChannel.processOps(int readyOps, SelectionKey key)` performs the I/O operations specified by `readyOps` bit mask, calling back to the application listener appropriately.

The goal is that once an application creates an async socket and opens a connection, the application does nothing else but send data and respond to the async socket callbacks.

Please note the callback sequence: `SelectorThread` ⇒ `AsyncChannel` ⇒ Application Listener. **The application listener is called back on the selector thread.** This means that while in the application listener, the selector thread is not watching for I/O events. So it is important for the application listener to handle the callback as quickly as possible. That said, there is a way to lessen the impact of a slow listener on priority socket channels.

Defining Selectors

Application developers may define one or more selectors to monitor channels in different ways. A selector is a thread encapsulating a `java.nio.channels.Selector` instance calling `Selector.select()` or variants of that method depending on the configured thread type. The thread types are:

- `ThreadType.BLOCKING`: selector thread calls `Selector.select()` and waits indefinitely or until there is a pending change to the key set. This type is most CPU friendly but also the slowest since the thread will be moved off core due to being blocked.

  The default eBus selector is blocking (but may be overridden by the application).
- `ThreadType.SPINNING`: select thread calls `Selector.selectNow()` repeatedly. This type is least CPU friendly since it will effectively take over a core but also the fastest to detect incoming bytes.

  Because a spinning thread dominates a core, it is recommended the selector thread be "pinned" to a specific core and that core isolated from the operating system. OpenHFT Thread Affinity API may be used to accomplish this.
- `ThreadType.SPINYIELD`: select thread calls `Selector.selectNow()` for a configured number of times. When that limit is reached the thread is parked using `java.util.concurrent.locks.LockSupport.park()`. When `park()` returns, selector thread begins spinning on `selectNow()` again for the configured limit.

  Like spinning, this type should have affinity for an isolated core.
- `ThreadType.SPINPARK`: like spin-yield, this type calls `Selector.selectNow()` for a configured number of times but parks for a configured nanosecond duration. When `park(long)` returns, selector thread begins spinning on `selectNow()` again.

  This type should have affinity for an isolated core.

The only way to define selectors is at start up using `-Dnet.sf.eBus.config.jsonFile=<file>`. An example selector configuration is:

```
"selectors" : [
    {
        "name" : "faster"      // required, must be unique.
        "type" : "spinning"    // required.
        "isDefault" : "false"  // optional, defaults to false.
        "priority" : 10        // optional, defaults to Thread.NORM_PRIORITY
        threadAffinity {       // optional, selector thread core affinity
            affinityType : CPU_ID // required, core selection type.
            cpuId : 7             // required for CPU_ID affinity type
            bind : true           // optional, defaults to false
            wholeCore : true      // optional, defaults to false
        }
    },
    {
        "name" : "slower"
        "type" : "spin+park"
        "isDefault" : "true"
        "priority" : 7
        "spinLimit" : 1000000
        "parkTime" : 500
    }
]
```

The default eBus selector is *not* created in this example since the configuration makes the "slower" spin+park selector the default.

The configured selectors may be retrieved either by name (`AsyncChannel.selector(String)`) or retrieving all known selectors (`AsyncChannel.selectors()`).

Please note that selector threads are *not* immediately started when configured. Rather selector threads are started when first referenced by an `AsyncChannel`. If no channel uses a particular selector thread, then that thread is not started.

Dynamic Selector Definition

eBus release 5.7.0 introduced the static method `AsyncChannel.createSelector(ENetConfigure.SelectorInfo info)` which creates a new selector thread based on the given selector configuration. This selector configuration is created using `ENetConfigure.SelectorInfoBuilder`. In turn a builder instance is obtained by calling `ENetConfigure.selectorBuilder()`.

Note that `info.isDefault()` *must* return false. This is because default selectors may only be defined during JVM initialization in the defined network configuration file.

Different Selectors for Different Channels

As pointed out above, application listeners are called out on the async channel's affiliated selector thread. This means that a slow listener can hold up other async channel I/O processing. But this problem can be lessened by segregating channels with slower listeners from priority channels with faster listeners by affiliating each type with a different `SelectorThread`.

The fast, high priority async channels use a spinning, core pinned selector thread; slow, low priority channels use a blocking, unpinned selector thread. This way the differing channels don't interfere with each other.

Selector Thread Affinity

eBus release 5.8.0 introduced thread affinity for selector threads. This affinity is created when adding `threadAffinity` property to the selector configuration. This works for both static configuration in the JSON file or dynamic Selector definition.

Thread affinity is recommended especially for selector threads configured as spinning since a thread doing hard spinning is not amenable to pre-emption. Also be sure to isolate the selected core from operating system use. Otherwise the spinning thread may still be pre-empted by the OS for its own particular use.

Please note: threadAffinity property is only supported in JSON configuration files and *not* in Java properties files.

See Appendix E: Configuring Thread Affinity for a detailed discussion on using thread affinity.

AsyncChannel Types

eBus networking API supports the following channels. The table matches the eBus async channel with its encapsulated Java NIO channel.

eBus Async Channel	Java NIO Channel
AsyncSocket	`SocketChannel`
AsyncServerSocket	`ServerSocketChannel`
AsyncSecureSocket	`SocketChannel` `SSLContext`
AsyncDatagramSocket	`DatagramChannel`
AsyncSecureDatagramSocket	`DatagramChannel`
AsyncMulticastSocket	`DatagramChannel`

See the eBus Javadoc API pages for complete information on how to use each eBus async channel type.

Appendix E: Configuring Thread Affinity

eBus release 5.8.0 introduced the ability to create thread affinity for dispatcher and selector threads. This feature is based on the OpenHFT Java Thread Affinity project. Class `net.sf.eBus.config.ThreadAffinityConfigure` defines thread affinity and is used by `net.sf.eBusx.util.ThreadAffinity`, directing its interaction with OpenHFT Java thread affinity.

ThreadAffinityConfigure consist of the following properties:

- `affinityType`: Required. Defines how core is acquired for the thread. There are five acquisition types as defined by enum `net.sf.eBus.config.ThreadAffinityConfigure.AffinityType`:

    1. `ANY_CORE`: Use `net.openhft.affinity.AffinityLock.acquireCore()` to assign any free core to thread.
    2. `ANY_CPU`: Use `AffinityLock.acquireLock()` to assign any free CPU to thread.
    3. `CPU_LAST_MINUS`: Use `AffinityLock.acquireLockLastMinus(int n)` to allocate a CPU from the end of the core set based on the given positive number. Requires property `lastMinusOffset` be set.
    4. `CPU_ID`: Use `AffinityLock.acquireLock(int cpuID)` to allocate a CPU with specified identifier to thread. Requires property `cpuId` be set.
    5. `CPU_STRATEGIES`: Use `AffinityLock.acquireLock(AffinityStrategies…)` to assign a CPU to thread. Requires property `cpuStrategies` be set.

        Please note that this type may not be used by itself or as an initial CPU acquisition type. Rather there must be previous CPU allocation to this (for example a previous dispatcher configuration using thread affinity) which the strategy then uses to allocate the next CPU. Attempts to use this acquisition type either by itself or as the first strategy will result in an error and no CPU allocated for the thread.

- `bind`: Optional, default value is `false`. If `true`, then bind current thread to allocated `AffinityLock`.
- `wholeCore`: Option, default value is `false`. If `true`, then bind current thread to allocated `AffinityLock` reserving the whole core. This property is used only with `bind` property is `true`.
- `cpuId`: Required when `affinityType` set to `CPU_ID`. Specifies the allocated CPU by its identifier.
- `cpuStrategies`: Required when `affinityType` set to `CPU_STRATEGIES`. Values are restricted to enum `net.openhft.affinity.AffinityStrategies`.

Note: strategy ordering is important. `AffinityStrategies.ANY` *must* appear as the last listed strategy. This allows any CPU to be selected in case none of the other strategies found an acceptable CPU.

Users should be familiar with the OpenHFT Java Thread Affinity library and how it works before using eBus thread affinity configuration. This includes configuring the operating system to isolate acquired CPUs from the operating system. This prevents the OS from pre-empting the thread from its assigned CPU which means the thread does not entirely own the CPU. That said, isolating too many CPUs from the OS can lead to a kernel panic. So using thread affinity is definitely an advanced software technique, requiring good understanding of how an OS functions.

The following example shows how to use thread affinity for eBus dispatcher threads and especially the CPU_STRATEGIES acquisition type.

```
"dispatchers" : [
    {
        "name" : "mdDispatcher"
        "numberThreads" : 1
        "runQueueType" : "spinning"
        "priority" : 9
        "quantum" : 10000
        "isDefault" : false
        "classes" : ["com.acme.trading.MDHandler"]
        threadAffinity {       // optional, selector thread core affinity
            affinityType : CPU_ID // required, core selection type.
            cpuId : 7             // required for CPU_ID affinity type
            bind : true           // optional, defaults to false
            wholeCore : true      // optional, defaults to false
    },
    {
        "name" : "orderDispatcher"
        "numberThreads" : 1
        "runQueueType" : "spinning"
        "priority" : 9
        "quantum" : 10000
        "isDefault" : false
        "classes" : ["com.acme.trading.OrderHandler"]
        threadAffinity {       // optional, selector thread core affinity
            affinityType : CPU_STRATEGIES // required, core selection type.
            cpuStrategies : [             // required for CPU_STRATEGIES affinity type
                    SAME_CORE, SAME_SOCKET, ANY // Note: ANY must be last strategy.
            ]
            bind : true           // optional, defaults to false
            wholeCore : true      // optional, defaults to false
    },
    {
        "name" : "defaultDispatcher"
        "numberThreads" : 8
        "runQueueType" : "blocking"
        "priority" : 4
        "quantum" : 100000
        "isDefault" : true
    }
]
```

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.