Java API: Tutorial

Version 1.4

Released: February 10, 2012

Preface

This document describes how to use the ION® Java® API Version 120 or higher to perform the following tasks:

- Set up the development environment.
- Write a simple publisher.
- Write a simple subscriber.
- Update records.
- Use transactions and functions.
- Write a trade subscriber.
- Write an Auto-Quote pricer.

Who Should Read This Document

This guide is intended for software developers interested in building custom ION Platform components.

It is assumed that the reader is familiar with the ION platform.

Related Documentation

- API documentation and samples suite
- *Auto-Quoting: User Guide and Reference*
- *Common Market Interface: User Guide and Reference*
- *Performance Meter: User Guide and Reference*
- *System Administrator Tool: User Guide and Reference*

Conventions Used in This Document

This document uses several conventions for special terms and actions, operating system-dependent commands, and paths.

Bold	Lowercase and mixed-case commands, command options, and flags that appear within text appear like **this**, in **bold** type. Graphical user interface elements and names of keys also appear like **this**, in **bold** type.
Italic	Variables, values you must provide, new terms, words, and phrases that are emphasized appear like *this*, in *italic* type.
`Monospace`	Commands, command options, and flags that appear on a separate line, paths and file names, code examples, output, and message text appear like `this`, in `monospace` type. Text strings you must type also appear like `this`, in `monospace` type.

Requirements

The requirements for this tutorial are:

- Running ION platform
- System Administrator Tool
- Runtime license
- ION Java API 120 or higher.
  This tutorial does not apply to older versions of the API.
- Java Development Kit (JDK®) 1.4.0 or higher.

Sections 7 and 8 also require:

- Common Market compliant gateway.
- Auto-Quoting enabled gateway.

For testing and instructive purposes, ION provides a market gateway simulator available for download from the ION Tracker Web site.

Contents

1. Setting up the Environment

ION® Platform components rely on a configuration file (`mkv.jinit`) to read the initial settings used to register the platform.

In the examples, the following `mkv.jinit` files are used:

mkv.jinit (Publisher)

```
mkv.cshost=localhost
mkv.csport=15000

mkv.component=PUB
mkv.listen=15826
mkv.user=user
mkv.pwd=pwd

mkv.debug=true
```

mkv.jinit (Subscriber)

```
mkv.cshost=localhost
mkv.csport=15000

mkv.component=SUB
mkv.listen=15926
mkv.user=user
mkv.pwd=pwd

mkv.debug=true
```

For further information and examples, refer to the API documentation and to the samples suite. Both are available on the ION Tracker Web site.

2. Writing a Simple Publisher

When you open the sample project for the first time, we see a basic skeleton for the publisher class:

```
public class Publisher
{
    public Publisher()
    {
    }

    public static void main(String[] args)
    {
    }
}
```

Starting and Registering the Component

The first step in writing any component with the Java® API is to create an instance of **`MkvQoS`** and complete the startup settings of the engine.

You use **`MkvQoS`** to define the initial set of listeners to be registered to the engine. You also use it to define the arguments received either from the command line or from the Process Manager.

To start the API, call the static method **`Mkv.start(MkvQoS qos)`**.

The **`Mkv`** class can have a single instance during the life of the component and can be retrieved using the static method **`getInstance()`**.

```
import com.iontrading.mkv.*;
import com.iontrading.mkv.events.*;
import com.iontrading.mkv.qos.*;
import com.iontrading.mkv.exceptions.*;
import com.iontrading.mkv.enums.*;


public class Publisher
{
    public Publisher()
    {
    }

    public static void main(String[] args)
    {
        Publisher pub = new Publisher();
        pub.startMkv(args);
    }

    private void startMkv(String[] args) {
        // create the initial configuration used to start the engine.
        MkvQoS qos = new MkvQoS();
        qos.setArgs(args);
        try {
            // Start the engine and get back the instance of Mkv (unique during the
            // life of a component).
            Mkv mkv = Mkv.start(qos);
        } catch (MkvException e) {
            // handle the exception
            e.printStackTrace();
        }
    }
}
```

Note: In this step, we have added the `import` lines at the top to reference the needed ION library packages.

Implementing MkvPlatformListener

Most components need to perform some initialization of data, publications, subscriptions, and so on.

This type of initialization is typically controlled by the API. This is because we want to wait until after the API has started and has registered the component before we start communicating with the platform.

To be notified when the API the status of our component changes and has registered, we must implement the `MkvPlatformListener` interface:

```
public class Publisher implements MkvPlatformListener
{
    …
    …
    private void startMkv(String[] args) {
        // create the initial configuration used to start the engine.
        MkvQoS qos = new MkvQoS();
        qos.setPlatformListeners(new MkvPlatformListener[] {this});
        qos.setArgs(args);
        try {
            // Start the engine and get back the instance of Mkv (unique during the
            // life of a component).
            Mkv mkv = Mkv.start(qos);
        } catch (MkvException e) {
            // handle the exception
            e.printStackTrace();
        }
    }
    …
    …
    public void onMain(MkvPlatformEvent mkvPlatformEvent) { }

    public void onComponent(MkvComponent component, boolean registered) { }

    public void onConnect(String component, boolean connected) { }
}
```

Note: In addition to the `onMain()` event we are interested in for now, the `MkvPlatformListener` interface also gives us `onComponent()` and `onConnect()` events. These events are typically useful for components that are designed to monitor the state of other components in the platform. Usually, components should not be interested in these events.

Handling onMain() Events

The `onMain()` event will be called several times during the initialization of our component, one for status change. The usual sequence is START -> REGISTER -> REGISTER_IDLE.

The `MkvPlatformEvent` object simulates an enumeration and uses an `intValue()` method to exploit the switch statement.

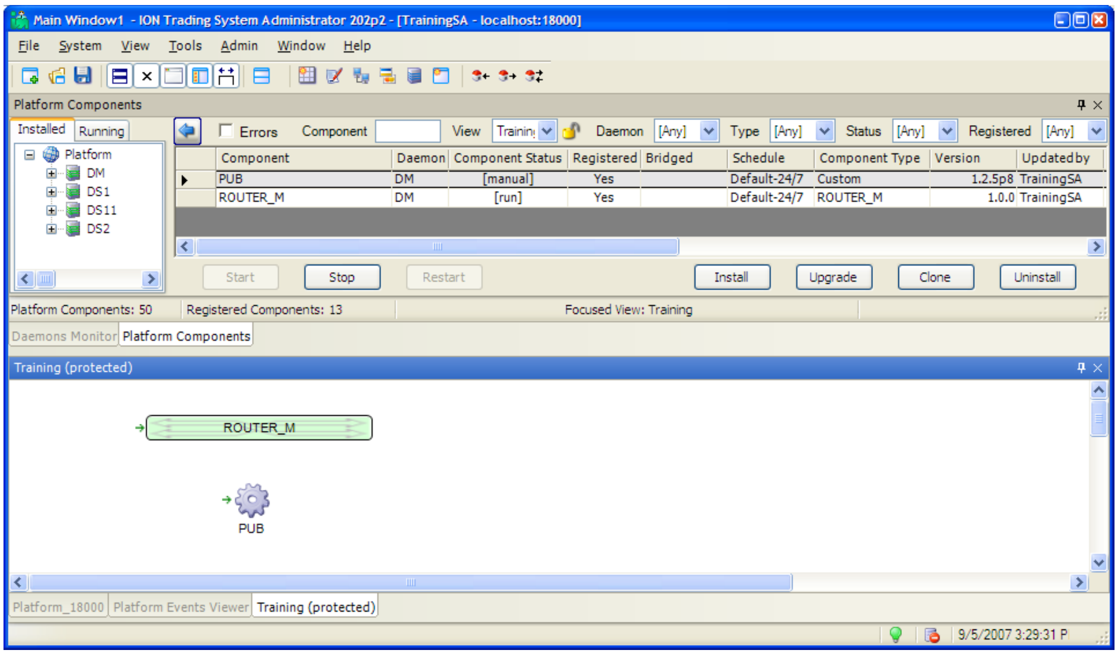
```
public void onMain(MkvPlatformEvent mkvPlatformEvent) {
    switch(mkvPlatformEvent.intValue()) {
        case MkvPlatformEvent.START_code:
            System.out.println("START");
            break;
        case MkvPlatformEvent.STOP_code:
            System.out.println("STOP");
            // Start the procedure fro gracesfully shutdwon the component
            // Save states, free resources, close connections ...
            break;
        case MkvPlatformEvent.REGISTER_IDLE_code:
            System.out.println("IDLE_REGISTER");
            break;
        case MkvPlatformEvent.REGISTER_code:
            System.out.println("REGISTER");
            break;
        }
}
```

For this example, we have inserted a `println()` statement for each case, so that we can see when each of these events occurs.

After the STOP event is fired, the engine can no longer interact with the platform. The application should therefore either start the API again or shut down gracefully.

At this point, we can run the Publisher and we should be able to see some output.

Additionally, we can open the System Administrator Tool at this point and we should see our PUBLISHER component running on the platform:


Main Window1 - ION Trading System Administrator 202p2 - [TrainingSA - localhost:18000]
File System View Tools Admin Window Help
Platform Components
Installed Running
Platform
DM
DS1
DS11
DS2
Errors Component View Trainin Daemon [Any] Type [Any] Status [Any] Registered [Any]
Start Stop Restart Install Upgrade Clone Uninstall
Daemons Monitor Platform Components
Training (protected)
ROUTER_M
PUB
Platform_18000 Platform Events Viewer Training (protected)
9/5/2007 3:29:31 P

Component	Daemon	Component Status	Registered	Bridged	Schedule	Component Type	Version	Updated by
PUB	DM	[manual]	Yes		Default-24/7	Custom	1.2.5p8	TrainingSA
ROUTER_M	DM	[run]	Yes		Default-24/7	ROUTER_M	1.0.0	TrainingSA

Platform Components: 50 Registered Components: 13 Focused View: Training

Publishing a Type

After the API is initialized correctly (no exception has been thrown), the application can start publishing objects. The following code shows how to publish a type to the platform.

The `publishType()` private method can be called after the initialization of the API within the START event.

```
private static final String TYPE_NAME = “MYPUB_Quote";
private static final String[] FIELD_NAMES = { "ID", "ASK", "BID", "QTY" };
private static final MkvFieldType[] FIELD_TYPES = {
            MkvFieldType.STR, MkvFieldType.REAL,
            MkvFieldType.REAL, MkvFieldType.REAL };

public void onMain(MkvPlatformEvent mkvPlatformEvent) {
    switch(mkvPlatformEvent.intValue()) {
        case MkvPlatformEvent.START_code:
            System.out.println("START");
            publishType();
            break;
    …
}
…
…
private void publishType() {
    try {
        MkvType type = new MkvType(TYPE_NAME, FIELD_NAMES, FIELD_TYPES);
        type.publish();
    } catch (MkvException e) {
        // handle the exception
        e.printStackTrace();
    }
}
```

Now we have published a type to the platform, using the naming convention `Source_TypeName`, with one string field and three real fields.

Publishing a Record

Once we have published a type, we can start publishing records and chains. For now, we will publish one record:

```
private String _recordName = "EUR.QUOTE.MYPUB.FGBLH3";
private MkvRecord record;

public void onMain(MkvPlatformEvent mkvPlatformEvent) {
    switch(mkvPlatformEvent.intValue()) {
        case MkvPlatformEvent.START_code:
            System.out.println("START");
            publishType();
            publishRecord();
            break;
    …
}

private void publishRecord() {
    try {
        MkvRecord rec = new MkvRecord(_recordName, TYPE_NAME);
        rec.publish();
        String fields[] = { "ID" };
        Object values[] = { "id" };
        rec.supply(fields, values);
        record = rec;
    } catch (MkvException e) {
        // handle the exception
        e.printStackTrace();
    }
}
```

In this step, we have published a record using the type we defined in the previous step (`MYPUB_Quote`), and using the standard record naming convention. We have then provided an initial supply of that record for the ID field (but all other fields will still be null and unsupplied).

Note: The record name we are using here is important because we will use this record later during the Auto-Quote exercise.

Supplying the Record

The next step is to supply some values for the record we have published:

```
public void onMain(MkvPlatformEvent mkvPlatformEvent) {
    switch(mkvPlatformEvent.intValue()) {
        case MkvPlatformEvent.START_code:
            System.out.println("START");
            publishType();
            publishRecord();
            // set the initial values for the record
            supplyRecord(99.9, 100.1, 5.0);
            break;
    …
}

private void supplyRecord(double ask, double bid, double qty) {
    String fields[] = {“ASK”, “BID”, “QTY”};
    Object values[] = {new Double(ask), new Double(bid), new Double(qty)};

    try {
        record.supply(fields, values);
        // ok, the record has been updated
    } catch (MkvException e) {
        e.printStackTrace();
        // handle the exception
    }
}
```

Publishing a Chain

Now that we have published and supplied a record, we can publish a chain and add the record to the chain:

```
private String _chainName = “EUR.QUOTE.MYPUB.QUOTE”;
private MkvChain chain;

public void onMain(MkvPlatformEvent mkvPlatformEvent) {
    switch(mkvPlatformEvent.intValue()) {
        case MkvPlatformEvent.START_code:
            System.out.println("START");
            publishType();
            publishRecord();
            // set the initial values for the record
            supplyRecord(99.9, 100.1, 5.0);
            chain = publishChain();
            break;
    …
}

private MkvChain publishChain() {
    try {
        MkvChain chain = new MkvChain(_chainName, TYPE_NAME);
        chain.publish();
        // add the record to the chain
        chain.add(_recordName);
        return(chain);
    } catch (MkvException e) {
        // handle the exception
        e.printStackTrace();
        return(null);
    }
}
```

Summary: Simple Publisher

So far, we have written a simple Publisher component. This component does the following:

- Starts and registers itself on the platform.
- Publishes a type.
- Publishes a record.
- Supplies the record.
- Publishes a chain.
- Appends the record to the chain.

3. Writing a Simple Subscriber

The initial project for the Subscriber component is just like the Publisher:

```
public class Subscriber
{
    public Subscriber()
    {
    }

    public static void main(String[] args)
    {
    }
}
```

Initializing the Component

Just like the Publisher, we must first initialize the API. The implementation of the **`MkvPlatformListener`** is the same of the Publisher case:

```
import ion.mkv.*;

public class Subscriber
{
    public Subscriber()
    {
    }

    public static void main(String[] args)
    {
        Subscriber sub = new Subscriber ();
        sub.startMkv(args);
    }

    private void startMkv(String[] args) {
        // create the initial configuration used to start the engine.
        MkvQoS qos = new MkvQoS();
        qos.setArgs(args);
        try {
            // Start the engine and get back the instance of Mkv (unique during the
            // life of a component).
            Mkv mkv = Mkv.start(qos);
        } catch (MkvException e) {
            // handle the exception
            e.printStackTrace();
        }
    }
}
```

Implementing MkvPublishListener

To receive the notification for the publication available on the platform, we need to register an **`MkvPublishListener`** to the engine. This can be set using the **`MkvQoS`** object:

```
public class Subscriber implements MkvPublishListener
{
    …
    private void startMkv(String[] args) {
        // create the initial configuration used to start the engine.
        MkvQoS qos = new MkvQoS();
        qos.setPublishListeners(new MkvPublishListener[] {this});
        qos.setArgs(args);
        try {
            // Start the engine and get back the instance of Mkv (unique during the
            // life of a component).
            Mkv mkv = Mkv.start(qos);
        } catch (MkvException e) {
            // handle the exception
            e.printStackTrace();
        }
    }
    …
    public void onPublish(MkvObject mkvObject, boolean start, boolean download) {}

    public void onPublishIdle(String component, boolean start) {}

    public void onSubscribe(MkvObject mkvObject) {}
}
```

The **`MkvPublishListener`** interface provides events that are usually exploited by consumers (publications) and others by providers (subscriptions). We are going to subscribe reacting to single publications for real time publications and reacting to the idle event for download publications. We are therefore going to implement our subscription logic in both the **`onPublish()`** and **`onPublishIdle()`** method.

Subscribing to a Chain

For our example Subscriber, the first thing we will do is subscribe to the chain that our Publisher component is publishing.

Since we are interested in receiving the updates related to the chain and the records belonging to the chain, we should also implement the **MkvChainListener** and **MkvRecordListener**.

In the example below, we have the Subscriber class implementing the two listeners' interfaces:

```
String chainName = "EUR.QUOTE.MYPUB.QUOTE";

public void onPublish(MkvObject mkvObject, boolean start, boolean download)
{
    if (download) {
        // wait for IDLE event to manage publication efficiently
        return;
    }
    if (start) {
        // a new object has been published
        // check if the component is interested in.
        if (mkvObject.getMkvObjectType().equals(MkvObjectType.CHAIN) &&
                mkvObject.getName().equals(chainName)) {
            try {
                ((MkvChain)mkvObject).subscribe(this);
                // the chain has been subscribed to
            } catch (MkvObjectNotAvailableException e) {
                // handle the exception
                // the object does not exists in the datadictionary
                e.printStackTrace();
            } catch (MkvConnectionException e) {
                e.printStackTrace();
                // the subscription failed because of a connection problem
            }
            } else {
                // the component is not interested in the just published object.
            }
        } else {
            // handle the un-publish.
            // Incase free the resources linked to the publication in case
            // it has been subscribed to.
        }
    }
}
public void onPublishIdle(String component, boolean start) {
    // query the internal database for the chain to subscribe
    MkvChain chain = Mkv.getInstance().getPublishManager().getMkvChain(chainName);
    if (start) {
        if (null == chain) {
            // the chain is not published jet
            return;
        }
        try {
            if (!chain.isSubscribed()) {
                chain.subscribe(this);
            }
        } catch (MkvException e) {
            e.printStackTrace();
        }
    } else {
        if (null == chain) {
            // handle the un-publish.
        }
    }
}
```

Note: Java API 120 requires passing a chain or record listener when subscribing to chains or records. For information about managing chains and records, see *Handling Chain Events*, on page 21.

In this case, we have hard-coded the name of the chain that we are interested in. In more typical real-world scenarios, the name of the chains or records that a subscriber might be interested in could be determined dynamically. For example, with a configuration file.

Note: The **`onPublish()`** method will be called both when the object is published and also when it is unpublished. we therefore need to check the **`start`** flag to determine which event calls the method.

Handling Chain Events

Once we have subscribed to a chain, there is a number of interesting chain events that we can listen for:

```
// listening for chain updates
public void onSupply(MkvChain mkvChain, String recordName, int position,
               MkvChainAction mkvChainAction) {
    System.out.println("Supplied chain " + mkvChain.getName());
    try {
        switch(mkvChainAction.intValue()) {
            case MkvChainAction.SET_code:
                // a snapshot of the chain is received, we wait for the IDLE_code
                // to process the chain
                System.out.println("Chain SET");
                break;
            case MkvChainAction.RESET_code:
                // the chain has been emptied.
                // NB. The argument record and position are useless in this case
                // TODO: Unsubscribe to all the recordrs.
                System.out.println("Chain RESET");
                break;
            case MkvChainAction.INSERT_code:
            case MkvChainAction.APPEND_code:
                // a record has been appended to the chain.
                System.out.println("Chain APPEND : " + recordName);
                subscribeToRecord (recordName);
                break;
            case MkvChainAction.DELETE_code:
                // a record has been removed from the chain.
                // TODO: Unsubscribe to the record.
                System.out.println("Chain DELETE : " + recordName);
                break;
            case MkvChainAction.IDLE_code:
                // the snapshot of the chain is completely received
                // NB. The argument record and position are useless in this case
                System.out.println("Chain IDLE");
                for(int i = 0; i < mkvChain.size(); i++) {
                    String _recordName = (String)mkvChain.get(i);
                    subscribeToRecord (_recordName);
                }
                break;
        }
    } catch (MkvException e) {
        // handle exceptions
        e.printStackTrace();
    }
}

private void subscribeToRecord(String recName) throws MkvException {
    MkvRecord rec =
        Mkv.getInstance().getPublishManager().getMkvRecord(recordName);
    if (!rec.isSubscribed()) {
        System.out.println("Subscribing " + recordName);
        rec.subscribe(this);
    }
}
```

In this example we have decided to subscribe to all Records produced by the Chain. In order for our subscriber to work as cleanly as possible, we also unsubscribe from Records when they are removed from the Chain.

Notes:

1. We would typically check the result of a call to `MkvRecord.subscribe()` for true or false, which will indicate success or an error.
2. We did not specify a list of fields in the record subscription. This means that we want to subscribe to the full record.

Handling Record Supply Events

To receive actual data, we can create a handler for the `onRecordSupply()` event:

```
// listen for partial record supplies (Implements MkvRecordListener.
public void onPartialUpdate(MkvRecord mkvRecord, MkvSupply mkvSupply,
           boolean isSnapshot) {
       //Deprecated
}

// listen only for full updates record supplies (Implements MkvRecordListener.
public void onFullUpdate(MkvRecord mkvRecord, MkvSupply mkvSupply,
           boolean isSnapshot) {
    try {
        String out = "Record " + mkvRecord.getName() + " : ";
        String[] fieldNames = MkvSupplyUtils.getFields(mkvRecord, mkvSupply);
        int cursor = mkvSupply.firstIndex();
        int counter = 0;
        while (cursor!=-1) {
            out +=  fieldNames[counter] +
                " {" + mkvSupply.getObject(cursor) + "} ";
            cursor = mkvSupply.nextIndex(cursor);
            counter++;
        }
        System.out.println("Updated {" + counter + "} fields : " + out);
    } catch (Exception e) {
        // handle exceptions
        e.printStackTrace();
    }
}
```

Summary: Simple Subscriber

We now have a simple Publisher and a simple Subscriber. Our Subscriber does the following:

- Starts and registers itself on the platform.
- Listens for a chain to be published.
- Subscribes to the chain.
- Listens to chain events.
- Subscribes to records in the chain.
- Listens for record supply events.

4. Updating Records

So far, our Publisher component is only capable of supplying one static set of values for our record. A more interesting case would be for our Publisher to supply continuously updating values for the record.

To make the sample more interesting, we can update the record on a timer base. In the example below, we use the **`MkvTimer`** helper. However, any thread mechanism can be used since the API is internally thread safe.

```
import java.util.*;


public void onMain(MkvPlatformEvent mkvPlatformEvent) {
    switch(mkvPlatformEvent.intValue()) {
        case MkvPlatformEvent.START_code:
            System.out.println("START");
            publishType();
            publishRecord();
            startSupplyTimer();
            break;
    …
}


private void startSupplyTimer()
{
    SupplyTimer timer = new SupplyTimer(_recordName);
    timer.start();
}

public class SupplyTimer extends MkvTimer
{
    private Random _rand = new Random();
    private String _fields[] = { "ASK", "BID" };
    private String _record;

    public SupplyTimer(String record)
    {
        super(“SupplyTimer”, 500, null);
        _record = record;
    }

    public boolean call(long t, Object ud)
    {
        double ask = 100.25 - _rand.nextDouble();
        double bid = 99.75 + _rand.nextDouble();
        supplyRecord(ask, bid, 5.0);
        return true;
    }
}
```

We now have our publisher continuously supplying new random data for the ASK and BID fields, updating every 500 milliseconds.

5. Using Transactions

We now have a publisher that is continuously updating the Bid and Ask prices for our record, but it is not updating the quantity. We can use a transaction to allow the Subscriber component to control when the quantity field should be updated.

Implementing a Transaction Handler

The first step is to implement a handler in the Publisher component for the specific transaction we want to support. To do this, we should implement `MkvTransactionListener` in the Publisher component. This is similar to what we previously did for the Subscriber component.

```
public class Publisher implements MkvPlatformListener, MkvTransactionListener
{
    ...

    private void startMkv(String[] args) {
        // create the initial configuration used to start the engine.
        MkvQoS qos = new MkvQoS();
        qos.setPlatformListeners(new MkvPlatformListener[] {this});
        qos.setTransactionListeners(new MkvTransactionlistener[] {this});
        qos.setArgs(args);
        try {
            // Start the engine and get back the instance of Mkv (unique during the
            // life of a component).
            Mkv mkv = Mkv.start(qos);
        } catch (MkvException e) {
            // handle the exception
            e.printStackTrace();
        }
    }

    ...

    public void onCall(MkvTransactionCallEvent event) {
        /**
         * @todo Implements the transaction support logic
         */
    }
}
```

We then need to implement code to handle the transaction inside the `onCall()` event:

```
public void onCall(MkvTransactionCallEvent event) {
    MkvRecord rec = event.getRecord();
    MkvSupply fieldsToBeChanged = event.getSupply();
    try {
        rec.supply(fieldsToBeChanged);
        event.setResult((byte)0, "OK");
    } catch (MkvException e) {
        try {
                event.setResult((byte)-1, "Failure:" + e);
            } catch (MkvException ee) {}
    }
}
```

This code will allow for any field to be updated and does not perform any validation or access checking. Typically, a publisher should check to make sure that the caller of the transaction is allowed to call the transaction and that the values supplied are valid.

Calling the Transaction

The next step is to call the transaction from the Subscriber component. We can create a timer class inside the Subscriber to continuously update the quantity field.

```
…
Import java.util.*;
private Timer myTimer = new Timer(true);
…
private void subscribeToRecord(String recName) throws MkvException {
    MkvRecord rec =
        Mkv.getInstance().getPublishManager().getMkvRecord(_recordName);
    if (!rec.isSubscribed()) {
        System.out.println("Subscribing " + _recordName);
        rec.subscribe(this);
        // create a task for each record to update
        myTimer.schedule(new UpdateTimer(rec), 0, 2000);
    }
}


class UpdateTimer extends TimerTask implements MkvTransactionCallListener
{
    private Random _rand = new Random();
    private String _fields[] = { "QTY" };
    private MkvRecord _record;
    private int _updateCount = 0;

    public UpdateTimer(MkvRecord record)
    {
        _record = record;
    }

    public void run()
    {
        double qty = 7.0 - 4 * _rand.nextDouble();
        System.out.println("Calling for transaction on " + record.getName() +
                " for QTY=" + qty);
        MkvSupplyBuilder builder = new MkvSupplyBuilder(_record);
        builder.setField("QTY", new Double(qty));
        try {
            _record.transaction(builder.getSupply(), this);
        } catch (Exception e) {
            // handle exception
            e.printStackTrace();
        }
    }

    /**
     *implement the MkvTransactionCallListener.onResult method
     */
    public void onResult(MkvTransactionCallEvent event, byte resCode,  String resStr)
    {
        System.out.println("Transaction res:" + resCode);
    }

}
```

Note: Here we have decided to start our timer immediately after subscribing to the record. In the Subscriber component we have several different calls to `MkvRecord.subscribe()`, depending on how the record is received in the `onChainSupply()` event. Make sure to start the timer for each of those cases.

In this case, our timer will randomly update the quantity field to a value between 5.0 and 9.0, continuously every 2 seconds.

Handling the Transaction Result

Note that the `UpdateTimer` object is either an `MkvTimer` and an `MkvTransactionCallListener` so that the object reference is passed to the `MkvRecord.transaction(…)`. This ensures that the response to the transaction is received.

6. Using Functions

Our Subscriber can now update the record’s quantity value using a transaction. We could also use a custom function. This can be preferable when component interaction requires the data owner (Publisher) to determine the actual values for the fields. Here, we implement a function to calculate and update the quantity field value based on several supplied inputs.

Implementing a Function Handler

In this example, we will create a method to publish the function and an inner class to implement the `MkvFunctionListener` which will manage the function calls:

```
Import com.iontrading.mkv.helper.*;

public void onMain(MkvPlatformEvent mkvPlatformEvent) {
    switch(mkvPlatformEvent.intValue()) {
        case MkvPlatformEvent.START_code:
            System.out.println("START");
            publishType();
            publishRecord();
            // set the initial values for the record
            supplyRecord(99.9, 100.1, 5.0);
            publishChain();
            publishFunction();
            break;
    …
}

private void publishFunction() {
    try {
        MkvFunction avg_function = new MkvFunction(
            "MYPUB_CalcQty", MkvFieldType.REAL, new String[] {"Arg1", "Arg2"},
            new MkvFieldType[] {MkvFieldType.REAL, MkvFieldType.REAL},
            "Calculate the mean value of two real values", new AvgFunctionHandler());
        avg_function.publish();
    } catch (MkvException e) {
        //handle exceptions
        e.printStackTrace();
    }
}

class AvgFunctionHandler implements MkvFunctionListener {
    public void onCall(MkvFunctionCallEvent mkvFunctionCallEvent) {
        try {
            MkvSupply argsWrapper = mkvFunctionCallEvent.getArgs();
            double arg0 = argsWrapper.getDouble(0);
            double arg1 = argsWrapper.getDouble(1);
            System.out.println("Called function Avg: arg1 {" +
                         arg0 + "} arg2 {" + arg1 + "}");
            double avg = (arg0 + arg1) / 2.0;
            System.out.println("Returning the result {" + avg + "}");

            String fields[] = {“QTY”};
            Object values[] = { new Double(avg)};
            record.supply(fields, values);

            mkvFunctionCallEvent.setResult(MkvSupplyFactory.create(avg));
        } catch (MkvException e) {
            // handle different scenarios of exception.
            try {
                mkvFunctionCallEvent.setError((byte)-1, "Exception {" + e + "}");
            } catch (Exception ee) {
                ee.printStackTrace();
            }
        }
    }
}
```

In this case, our Function `CalcQty` calculates and updates the quantity field to the midpoint value between the two supplied inputs. The function also uses the new quantity value as the return value.

Notes:

1. We have avoided most of the work typically done in a function, such as validating arguments, checking bounds, and so on. In a real-world scenario, we would typically examine the parameters passed in to ensure that they are compatible.
2. Because functions are platform objects, they should be published to the platform. This operation is performed in the **`publishFunction`** method, called by the START event.

In this example, we have published a function called `CalcQty`, with a return value of REAL, and with two input parameters `QTY1` and `QTY2`, both of type REAL. Finally, we have provided a user-friendly description for the function.

Calling the Function

From the Subscriber side, we can replace the code inside our update timer to call the function instead of calling the transaction:

```
class UpdateTimer extends TimerTask implements MkvFunctionCallListener

    private MkvFieldType[] _types = { MkvFieldType.REAL, MkvFieldType.REAL };
    private MkvFunction meanValueFunction = null;
    private Random _rand = new Random();

    public void run()
    {
        lazyLoadFunction();
        if (meanValueFunction==null) {
            System.out.println("The function is not available!");
        } else {
            double qty1 = 7.0 - 4 * _rand.nextDouble();
            double qty1 = 7.0 - 4 * _rand.nextDouble();

            MkvSupply args = MkvSupplyFactory.create(new Object[] {
                     new Double(qty1), new Double(qty2)});
            try {
                 meanValueFunction.call(args, this);
                System.out.println("Function Avg called succesfully");
            } catch (MkvException e) {
                e.printStackTrace();
            }
        }
    }

    private void lazyLoadFunction() {
        if (meanValueFunction==null || !meanValueFunction.isValid()) {
            meanValueFunction =
                Mkv.getInstance().getPublishManager().getMkvFunction("MYPUB_CalcQty");
        }
    }

    public void onError(MkvFunctionCallEvent event, byte errCode, String errStr) {
        // handle function error
    }

    public void onResult(MkvFunctionCallEvent event, MkvSupply result) {
        try {
            System.out.println("Function res {" + result.getDouble(0) + "}");
        } catch (MkvException e) {
            e.printStackTrace();
        }
    }
}
```

Handling the Function Result

Handling the result of the function is very similar to handling the result of the transaction. In this case, the inner class `UpdateTime` will not implement the `MkvTransactionCallListener` any more. Instead, it will implement the `MkvFunctionCallListener` interface to get back the result of the function call.

7. Writing a Trade Subscriber

The goal of this exercise is to gain some initial exposure to real-world data in the ION Platform by writing a simple trade Subscriber component. We will work from the Subscriber component written previously and adapt it to subscribe to Common Market trade records published by actual gateway components on the ION Platform. For the purposes of this exercise, we will test our Subscriber component by connecting it to the Mock gateway that is running on the test platform.

The example shown in this section subscribes to the TRADE chain using the mechanism already described (`MkvChainListener` + `MkvRecordListener`). There is a different mechanism for dealing with chains that exploit the new “permanent subscription” features of the API introduced with the Version 120. Advanced samples of permanent subscriptions are available in the standard samples suite downloadable from the ION Tracker Web site.

Implementing MkvRecordListener

We are going to create an object to subscribe to a CM_TRADE record and receive the updates. The class will subscribe to the CM_TRADE record in the constructor and will cache the field IDs to avoid looking for values by field name.

```
class TradeRecordListener implements MkvRecordListener
{
    private String[] _fields = new String[]{ "InstrumentId", "Qty", "Price", "VerbStr"
};
    private int[] _fieldsIdxs = new int[4];

    private static final int INSTR_ID = 0;
    private static final int QTY= 1;
    private static final int PRICE = 2;
    private static final int VERB = 3;

    private MkvRecord rec;

    public TradeRecordListener(String recordName) throws MkvException
    {
        System.out.println("Subscribe to Trade record : " + recordName);
        rec = Mkv.getInstance().getPublishManager().getMkvRecord(recordName);
        rec.subscribe(_fields, this);
         _fieldsIdxs[INSTR_ID] = rec.getMkvType().getFieldIndex(_fields[INSTR_ID]);
         _fieldsIdxs[QTY] = rec.getMkvType().getFieldIndex(_fields[QTY]);
         _fieldsIdxs[PRICE] = rec.getMkvType().getFieldIndex(_fields[PRICE]);
         _fieldsIdxs[VERB] = rec.getMkvType().getFieldIndex(_fields[VERB]);
    }

    public void onPartialUpdate(MkvRecord mkvRecord, MkvSupply mkvSupply,
                   boolean isSnapshot) {
        // not interested
    }

    public void onFullUpdate(MkvRecord mkvRecord, MkvSupply mkvSupply,
                   boolean isSnapshot)
    {
         try {
             String instrId = mkvRecord.getSupply().getString(_fieldsIdxs[INSTR_ID]);
             String verb = mkvRecord.getSupply().getString(_fieldsIdxs[VERB]);
             double qty = mkvRecord.getSupply().getDouble(_fieldsIdxs[QTY]);
             double price = mkvRecord.getSupply().getDouble(_fieldsIdxs[PRICE]);
             System.out.println("TRADE : " + instrId + " " + verb + " : " + qty + " @ "
+ price);
         } catch (Exception e) {
             // handle the exception
             e.printStackTrace();
         }
    }

    public void unsubscribe() throws MkvException
    {
        rec.unsubscribe(this);
    }
}
```

Our record listener is specific to CM_TRADE publications, and subscribes to four fields which are part of the standard Common Market definition for trade records:

- `InstrumentId`
- `Qty`
- `Price`
- `VerbStr`

Subscribing to the Chain

The listeners for the single CM_TRADE records will be instantiated (so the subscription will be performed) by an object responsible for subscribing the TRADE chain and receiving the updates of the list of trades belonging to the chain. The object will also cache the references to the trade listener objects.

Since we delegated the logic related to single records to a specific class managing the chain is quite straightforward:

```
public class TradeChainListener implements MkvChainListener
{
    private Map trades = new HashMap();

    public TradeChainListener(MkvChain chain)
    {
        System.out.println("Subscribe to Trade chain : " + chain.getName());
        try { chain.subscribe(this); } catch(MkvException e) {}
    }

    private void listenTrade(String recordName) {
        TradeRecordListener tradeListener = null;
        try {
                tradeListener = new TradeRecordListener(recordName);
             }
                catch (MkvException e) {
             }
        trades.put(recordName, tradeListener);
    }

    public void onSupply(MkvChain chain, String record, int pos,
                        MkvChainAction action) {
                    switch(action.intValue()) {
                 case MkvChainAction.SET_code:
                     for(int i = 0; i < chain.size(); i++) {
                         listenTrade((String)chain.get(i));
                     }
                     break;
                 case MkvChainAction.INSERT_code:
                 case MkvChainAction.APPEND_code:
                 {
                       listenTrade(record);
                     break;
                 }
             }
    }

}
```

Subscribing to CM_TRADE Chains

The final step in implementing our trade Subscriber component is to recognize the chains we are interested in and to subscribe to those chains. To do this, we make some modifications to the `onPublish()` event handler in our Subscriber component to create the instance of `TradeChainListener` reacting to the TRADE chain publication.

In a real-world scenario, we may also want to handle the unpublish event, in which case we should unsubscribe the chain listener and remove it from the list of listeners.

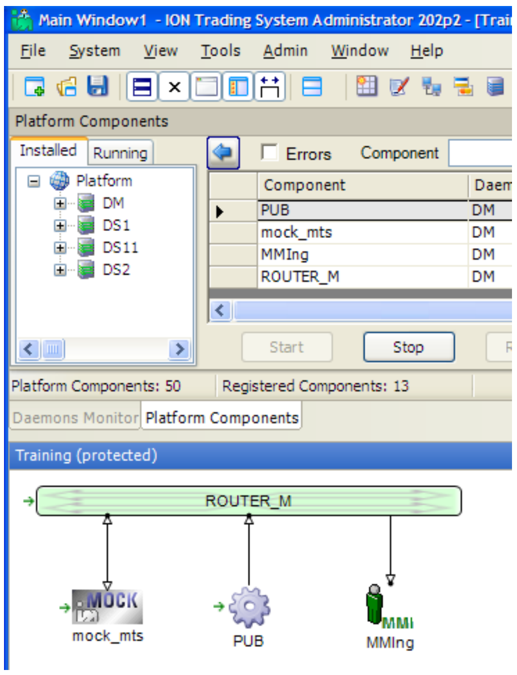
8. Writing an Auto-Quote Pricer

We have already written a Publisher component that is compatible with the Auto-Quote capabilities of the ION gateways. The only requirements for this are that our publisher must publish records corresponding to the standard naming conventions in the ION Platform, and we must make sure our Instrument ID can be matched to the Instrument ID we want to price inside a gateway.

Once we have our publisher working, the only remaining work to do for this exercise is to configure the MOCK gateway to recognize our pricing component. This is done by editing the `mkv.init` file in the directory where the MOCK is located:

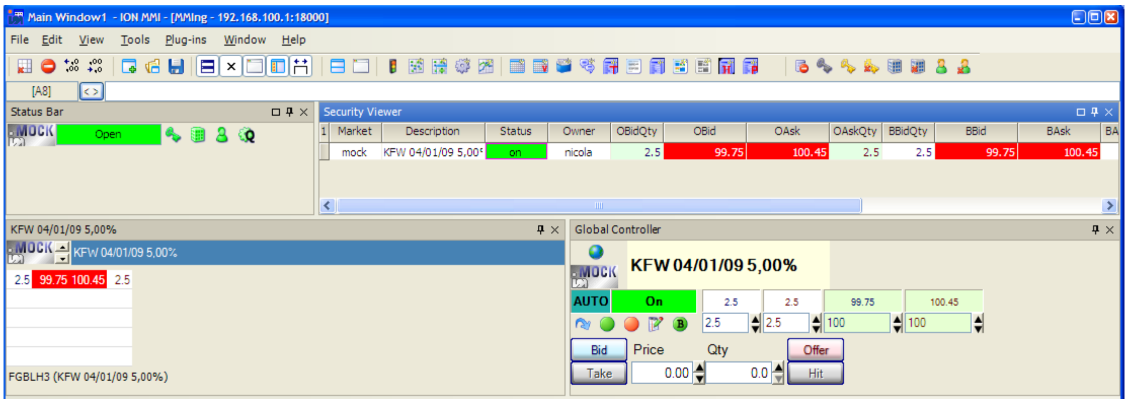
```
set MKVQUOTE_RECORDS       EUR.QUOTE.MYPUB.
set MKVQUOTE_BYRECORD      1
set MKVQUOTEMAP_BIDPRICE   BID
set MKVQUOTEMAP_ASKPRICE   ASK
```

After editing the configuration file for the MOCK gateway, the process must be killed and then MOCK must be restarted. Once MOCK has been restarted and we have our Publisher component running, they should be connected with "InOut" connections to the ROUTER component:



Component	Daem
PUB	DM
mock_mts	DM
MMIng	DM
ROUTER_M	DM

Once all components are connected correctly, we can load the user interface, load the appropriate instrument (FGBLH3) into the quote panel, and switch on Auto-Quoting. It might also be necessary to change the quote mode to Bid/Ask. This is done by right-clicking the quote in the quote panel, then selecting **Change Quote -> BidAsk** from the pop-up menu.



Main Window1 - ION MMI - [MMIng - 192.168.100.1:18000]

1	Market	Description	Status	Owner	OBidQty	OBid	OAsk	OAskQty	BBidQty	BBid	BAsk	BA
	mock	KFW 04/01/09 5,00'	on	nicola	2.5	99.75	100.45	2.5	2.5	99.75	100.45	

KFW 04/01/09 5,00%

2.5 99.75 100.45 2.5

FGBLH3 (KFW 04/01/09 5,00%)

KFW 04/01/09 5,00%

AUTO On 2.5 2.5 99.75 100.45

2.5 2.5 100 100

Bid Price Qty Offer

Take 0.00 0.0 Hit

9. Writing a Message Queue Publisher

Message queues provide point-to-point connectivity between two specific components. This avoids the unnecessary distribution of messages to other platform components.

Message queues are the ideal form of information distribution where two components exchange one-time data using a private channel.

In this scenario, the message queue model is generally more efficient than a publish / subscribe model. This is particularly important where large volumes of *use once* data are exchanged between two components. For example: downloading static data, fielding incoming trade execution (order fills), providing STP services, and sourcing historical / tick data.

To set up a message queue, you must create a configuration object and pass it to an MkvMQManager instance. This will create an MkvMQ object instance allowing you to write records to the message queue.

The publisher can dispose of a message queue object at any time by calling the `close` method with a timeout. The subscriber will be able to consume the remaining records in the queue until the specified time elapses.

The full code for the example is available in the standard samples suite which you can download from the ION Tracker Web site.

Creating a Message Queue

Below we show the code for the Function listener inner class in the example, the onCall handler publishes, populates, and releases a Message Queue object.

```
private class FunctionHandler implements MkvFunctionListener {
    final MkvType mkvType;
    FunctionHandler()
    {
        MkvType _mkvType = null;
        /* create MkvType for dynamically created message queue */
        try {
            _mkvType = new MkvType("TRADETYPE",
                    new String [] { "Code", "Price", "Qty", "VerbStr" },
                    new MkvFieldType[] { MkvFieldType.STR, MkvFieldType.REAL,
                    MkvFieldType.REAL, MkvFieldType.STR } );
        }
        catch (MkvException e) { }
        finally {
            mkvType = _mkvType;
        }
    }
    public void onCall(MkvFunctionCallEvent mkvFunctionCallEvent) {
        try {
            MkvSupply argsWrapper = mkvFunctionCallEvent.getArgs();
            StringBuffer mqName = new StringBuffer();
            /* generate a random name for a dynamically created message queue */
            mqName.append("ANY.SPLITRES.").append(SOURCE).
                append(".TEMP_").append(Math.abs(new Random().nextInt()));
            /* retrieve input string */
            String argument = argsWrapper.getString(0);
            /* create a message queue with the random generated name */
            MkvMQ mkvMQ = mkv.getMQManager().create(mqName.toString(), mkvType,
                new MkvMQConf());
            long seed = argument.hashCode();
            Random r = new Random(seed);
            DecimalFormat df = new DecimalFormat("0000000000");
            for (int i = 0; i < ((seed % 10) + 1); ++i)
                mkvMQ.put(MkvSupplyFactory.create(new Object[] {
                        "CH" + new String(df.format(Math.abs(r.nextInt()) % 10000)),
                        new Double(90. + 2 * r.nextDouble()),
                        new Double(10000 * ( + 1)),
                        new String((r.nextInt() % 2 == 0) ? "Buy" : "Sell") }));
            /* close the queue with a 10 second timeout */
            mkvMQ.flush();
            mkvMQ.close(10);
            /* the name of the queue is returned as the function call result
             * so that the caller can subscribe to it */
            mkvFunctionCallEvent.setResult(MkvSupplyFactory.create(mqName.toString()));
        } catch (MkvException e) {
            try {
                mkvFunctionCallEvent.setError((byte)-1,
                    "Unexpected exception {" + e + "}");
            } catch (Exception ee) {
                ee.printStackTrace();
            }
        }
    }
}
```

10. Writing a Message Queue Subscriber

The example shown in this section is designed to work together with the Message Queue Publisher in the previous section.

The subscriber invokes the function published by the publisher, thus obtaining the name of a message queue. It then subscribes to the queue and retrieves the records the publisher delivered to it.

Subscribing to a message queue entails setting up an MkvMQListener instance and associating it to an MkvMQSubscribe object. The subscription is started by invoking the subscribe method of the MkvMQSubscribe instance.

The full code for the example is available in the standard samples suite that can be downloaded from the ION Tracker Web site.

Subscribing to a Message Queue

Below we show the code for the FunctionListener inner class in the example. This class implements the subscription to the message queue.

```
private class FunctionListener implements MkvFunctionCallListener {
    public void onError(MkvFunctionCallEvent mkvFunctionCallEvent,
            byte errCode, String errStr) {

        System.out.println("Error result for GetLastTradesByTrader function: " +
                "code {" + errCode + "} message {" + errStr + "}");
    }

    public void onResult(
            MkvFunctionCallEvent mkvFunctionCallEvent, MkvSupply result) {
        /* on servicing the function call,
         * MessageQueuePublisher publishes some data onto a dynamically created queue,
         * whose name it returns as the function result */
        try {
            String resultMQ = result.getString(0);
            System.out.println(
                "Publisher is returning result for function on message queue {" +
                resultMQ + "}");
            /* subscribe the queue */
            MkvMQSubscribe mkvMQSubscribe =
                Mkv.getInstance().getMQManager().createSubscription(resultMQ,
                        /* set up a listener for data supplied to the queue */
                        new MQSubscribeListener(resultMQ),
                        /* use defaults for subscription */
                        new MkvMQSubscribeConf());
            mkvMQSubscribe.subscribe();
        } catch (MkvException e) {
            e.printStackTrace();
        }
    }
}
```

Handling Message Queue events

Below we show the code for the MQListener inner class in the example. This class implements the MkvMQSubscribeListener interface.

```
private class MQSubscribeListener implements MkvMQSubscribeListener {
    final String queueName;
    public MQSubscribeListener(String queueName) {
        this.queueName = queueName;
    }
    /* handler for message queue related events */
    public void onUpdate(MkvMQSubscribeEvent event) {
        try {
            switch (event.getSubscribeAction().intValue()) {
            case MkvMQSubscribeAction.SUPPLY_code:
                /* a record was received from the queue */
                System.out.println("Reading result from queue {" + queueName +
                    "} -> Code: {" +  event.getSupply().getString(0) +
                    "} Price: {" + event.getSupply().getDouble(1) +
                    "} Qty: {" + event.getSupply().getDouble(2) +
                    "} VerbStr: {" + event.getSupply().getString(3));

                break;
            case MkvMQSubscribeAction.CLOSE_code:
                /* the publisher has closed the queue -
                 * no more data will be received from it */
                System.out.println("Ephemeral message queue {" + queueName +
                    "} has been closed");
            default:
                ;
            }
        }
        catch(MkvException e) {
            e.printStackTrace();
        }

    }
}
```

11. Using Custom Statistics

From Java API Version 130p1, custom components can contribute to the system with their own performance indicators.

Custom components can define the type and characteristics of the new indicators and provide the values for them at runtime, when needed. Custom statistics are also synchronized with the standard statistics.

The custom indicators produced by custom components are reported to the Daemon Master and then processed by the ION Performance Meter Tool (PerfMeter®) in the same way as standard indicators.

When you use PerfMeter Version 200 or later with the System Administrator Tool Version 220 or later, the custom indicators are available in the PerfMeter dashboards. For more information refer to the *Performance Meter: User Guide and Reference*.

Use Cases

Using the Java API, a custom component can contribute to the PerfMeter system in the following ways:

- **Extending existing tables:**
  The custom component can extend any existing dashboard, providing values for additional columns.
- **Defining new tables:**
  The custom component can create new tables to report a set of indicators in the same new category that is business dependent.
- **Creating virtual entries in existing tables:**
  The custom component can provide standard indicators for virtual entities.

In each case, the first step is to define the names of the new indicators along with their types. The supported types are the usual field types defined on the ION Platform.

The custom component should also give a symbolic name to the table of custom indicators it is going to publish.

The final step is to provide the actual values for the indicators when the Java API requires them, reacting to explicit events from the API.

Extending Existing Tables

This is the typical use case. A custom component needs to provide new performance indicators describing its business logic, along with the standard information already provided by the API.

For example, a custom component can define two new indicators:

- A concept of *available processing bandwidth*
- The number of trades processed in each time interval.

Defining the Metadata:

The following snippet creates a new `TradeProcessor` table with two indicators:

- `Bandwidth` (an integer)
- `NumTrades` (a real number)

The new table must contain a field (`COMPONENTS_KEY_FIELD` in this case) matching the key field for the standard table it is extending.

Finally, the snippet registers a `TradeProcessorListener` object that will be invoked by the API to retrieve current values for the statistics.

```
MkvType customType = new MkvType("TradeProcessorType",
   new String[] { MkvCustomStatsManager.COMPONENTS_KEY_FIELD,
          "NumTrades", "Bandwidth" },
   new MkvFieldType[] { MkvFieldType.STR,
             MkvFieldType.INT, MkvFieldType.REAL });

Mkv.getInstance().getCustomStatsManager()
  .createTable("TradeProcessor", customType, new TradeProcessorListener())
  .extendTable(MkvCustomStatsManager.COMPONENTS_TABLE);
```

Reporting the Custom Stats:

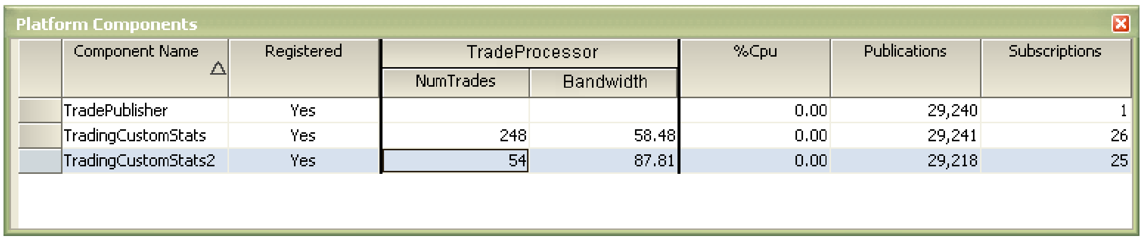
The `TradeProcessorListener` object reports the values of the Custom Stats to the Daemon. It will be invoked to send out custom statistics in sync with the standard statistics.

```
static class TradeProcessorListener implements MkvCustomStatsListener
{
  public void produceStats(MkvCustomStatsTable table) {
    MkvSupplyBuilder builder = new MkvSupplyBuilder(table.getType());

    builder.setField(table.getType().
           getFieldIndex(MkvCustomStatsManager.COMPONENTS_KEY_FIELD),
           Mkv.getInstance().getProperties().getComponentName());

    // Compute custom indicator values here
    builder.setField(table.getType().getFieldIndex("NumTrades"), tradesSoFar);
    builder.setField(table.getType().getFieldIndex("Bandwidth"), bandwidth);

    table.send(builder.getSupply());
  }
}
```

Browsing the Custom Stats using the System Administrator Tool:

Extensions to the existing tables will be available both from Real Time and Historical dashboards. Custom Stats can be shown in any grid using the **Options** dialog.

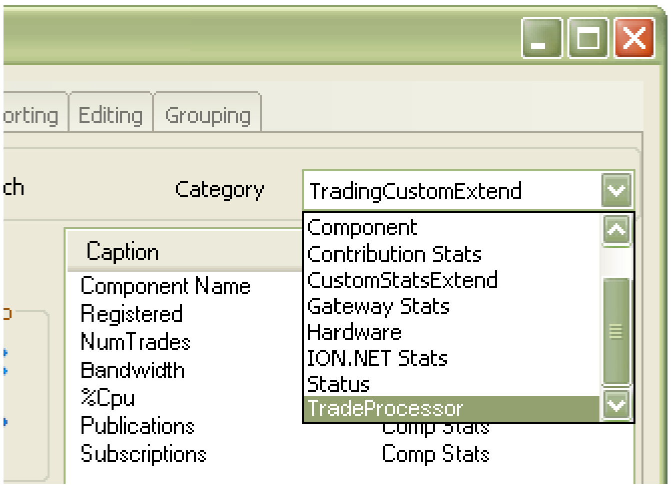
Custom statistics generated by the code in the previous section will be displayed in a Real Time dashboard as follows:



Platform Components

Component Name	Registered	TradeProcessor		%Cpu	Publications	Subscriptions
		NumTrades	Bandwidth			
TradePublisher	Yes			0.00	29,240	1
TradingCustomStats	Yes	248	58.48	0.00	29,241	26
TradingCustomStats2	Yes	54	87.81	0.00	29,218	25

The name of the newly created Custom Stats table is available in the **Category** drop-down list in the **Options** window:



For more information about using PerfMeter dashboards, refer to the *Performance Meter: User Guide and Reference*.

External Tables

In this use case, a custom component needs to provide additional statistical dimensions about data already available in another table. External tables can be navigated using the System Administrator Tool from the component statistics tables.

For example, a GUI containing multiple data grids may want to show two per-grid indicators: `Rows` and `Redraw`.

Defining the Metadata:

The following code snippet defines the new Custom Stats. In addition to the steps covered in the previous use case, the snippet adds an extra key field (`GridName`). This field will be used by the System Administrator Tool to show disaggregated per-grid statistics.

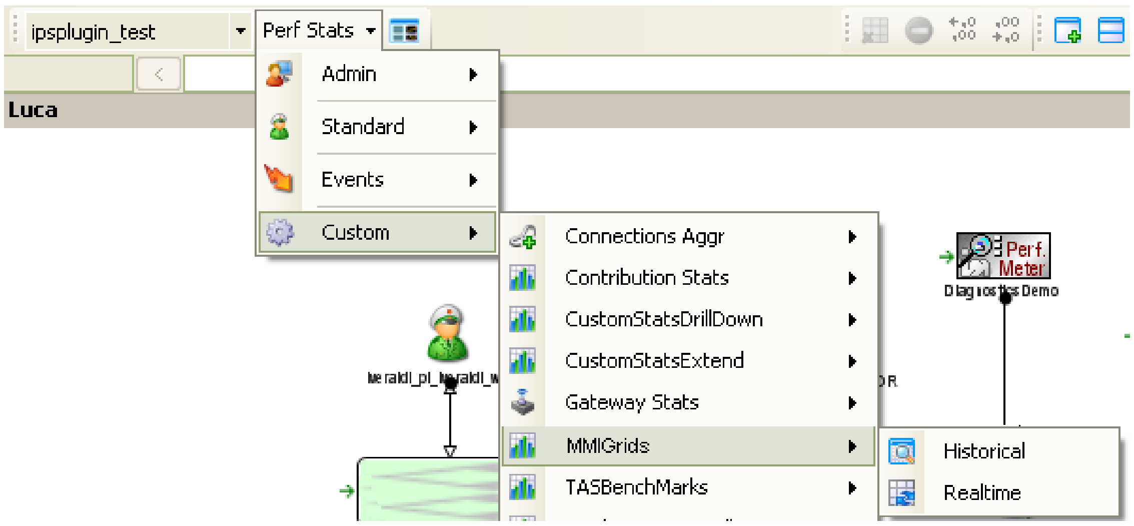
```
MkvType customType = new MkvType("MMIGridsType",
    new String[] {
        MkvCustomStatsManager.COMPONENTS_KEY_FIELD,
        "GridName",
        "Rows", "Redraw" },
    new MkvFieldType[] { MkvFieldType.STR,
        MkvFieldType.STR,
        MkvFieldType.INT, MkvFieldType.INT });

Mkv.getInstance().getCustomStatsManager()
    .createTable("MMIGrids", customType, new MMIGridsListener())
    .extendTable(MkvCustomStatsManager.COMPONENTS_TABLE)
    .addKeyFields(new String[] { "GridName" });
```

Reporting the Custom Stats:

The simple code snapshot below reports the values of Custom Stats.
It creates multiple lines, all with the same `COMPONENTS_TABLE` primary key (and thus all the external table lines will correspond to the same component) but differing by their `GridName`:

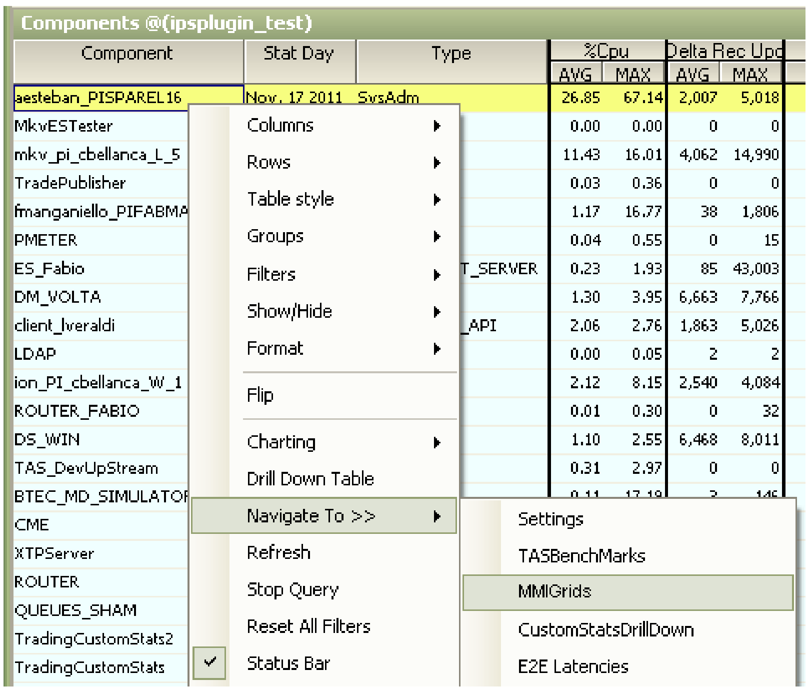
```
static class MMIGridsListener implements MkvCustomStatsListener
{
    public void produceStats(MkvCustomStatsTable table)
    {
        MkvSupplyBuilder builder = new MkvSupplyBuilder(table.getType());

        builder.setField(
            table.getType().getFieldIndex(MkvCustomStatsManager.COMPONENTS_KEY_FIELD),
            Mkv.getInstance().getProperties().getComponentName());

        for(int i=0; i<marketMonitors.size(); ++i) {
            builder.setField(
                table.getType().getFieldIndex("GridName"),
                marketMonitors[i].getName());
            // Compute custom indicator values here
            builder.setField(table.getType().getFieldIndex("Rows"), rows);
            builder.setField(table.getType().getFieldIndex("Redraw"), redraw);
        }

        table.send(builder.getSupply());
    }
}
```

Browsing External Tables using the System Administrator Tool:



External tables are available both from Real Time and Historical dashboards.

The new drill-down tables can also be accessed using the Navigation feature:



Components @(ipsplugin_test)

Component	Stat Day	Type	%Cpu		Delta Rec Upd	
			AVG	MAX	AVG	MAX
aesteban_PISPAREL16	Nov. 17 2011	SysAdm	26.85	67.14	2,007	5,018
MkvESTester			0.00	0.00	0	0
mkv_pi_cbellanca_L_5			11.43	16.01	4,062	14,990
TradePublisher			0.03	0.36	0	0
fmanganiello_PIFABMA			1.17	16.77	38	1,806
PMETER			0.04	0.55	0	15
ES_Fabio		T_SERVER	0.23	1.93	85	43,003
DM_VOLTA			1.30	3.95	6,663	7,766
client_lveraldi		_API	2.06	2.76	1,863	5,026
LDAP			0.00	0.05	2	2
ion_PI_cbellanca_W_1			2.12	8.15	2,540	4,084
ROUTER_FABIO			0.01	0.30	0	32
DS_WIN			1.10	2.55	6,468	8,011
TAS_DevUpStream			0.31	2.97	0	0
BTEC_MD_SIMULATOR			0.11	17.19	[illegible]	146
CME						
XTPServer						
ROUTER						
QUEUES_SHAM						
TradingCustomStats2						
TradingCustomStats						

For more information about using the PerfMeter dashboards, refer to the *Performance Meter: User Guide and Reference*.

Virtual Records

In this use case, a custom component works as a proxy and needs to publish information about components / connections / hosts that are not real ION Platform entities but should be virtualized.

For example, consider a component working as a proxy between the ION Platform and a legacy world, handling a set of other applications that are not registered to the ION Platform and which are also running on other machines.

This component may want to publish the standard statistics (or a subset of them) for the hosts that are not managed by ION Daemon instances. For example, `NumProcs`, `CpuPct`, `MemPct`.

Defining the Metadata:

The following snippet defines the Custom Stats. Except for the different records and standard table, it is the same as the snippet for the non-virtual use case:

```
MkvType customType = new MkvType("HInfoType",
   new String[] {
        MkvCustomStatsManager.HOSTS_KEY_FIELD,
        "NumProcs",
        "CpuPct", "MemPct" },
   new MkvFieldType[] {
        MkvFieldType.STR,
        MkvFieldType.INT,
        MkvFieldType.REAL, MkvFieldType.REAL });

Mkv.getInstance().getCustomStatsManager()
    .createTable(MkvCustomStatsManager.HOSTS_TABLE, customType, new HostsListener())
    .extendTable(MkvCustomStatsManager.HOSTS_TABLE);
```

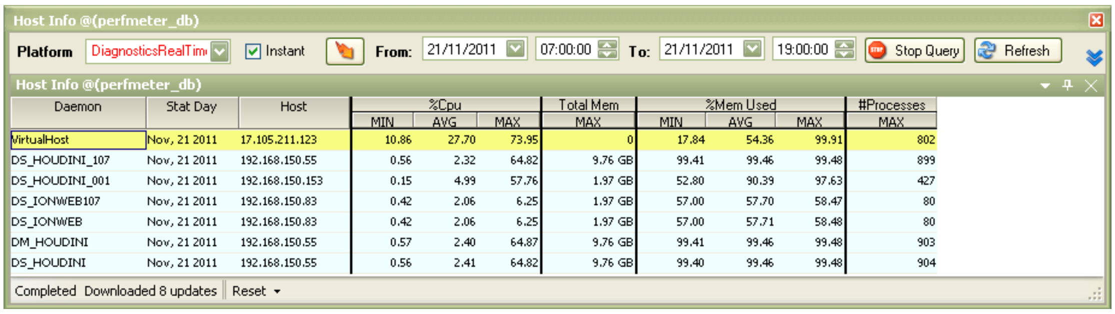
Reporting the Custom Stats:

The following snapshot contributes the actual statistics. Again, it is nearly the same as the snippet for the non-virtual case but the primary key is explicitly set to a specific value corresponding to the IP for the non-Daemon managed host (`123.123.123.123`) instead of looking up the name for the current machine:

```
/*
 * Produce the Custom Stats when the Engine requires
 */
static class HostsListener implements MkvCustomStatsListener
{
    public void produceStats(MkvCustomStatsTable table) {
        MkvSupplyBuilder builder = new MkvSupplyBuilder(table.getType());

        builder.setField(
            table.getType().getFieldIndex(MkvCustomStatsManager.HOSTS_KEY_FIELD),
           "123.123.123.123");

        // Compute custom indicator values here

        builder.setField(table.getType().getFieldIndex("NumProcs"), numprocs);
        builder.setField(table.getType().getFieldIndex("CpuPct"), cpu);
        builder.setField(table.getType().getFieldIndex("MemPct"), mem);

        table.send(builder.getSupply());
    }
}
```

Browsing the Custom Stats using the System Administrator Tool:

Virtual Records are shown in the corresponding table, in the same way as any non-virtual information:


Host Info @(perfmeter_db)
Platform
DiagnosticsRealTim
Instant
From: 21/11/2011 07:00:00
To: 21/11/2011 19:00:00
Stop Query
Refresh
Host Info @(perfmeter_db)

Daemon	Stat Day	Host	%Cpu			Total Mem	%Mem Used			#Processes
			MIN	AVG	MAX	MAX	MIN	AVG	MAX	MAX
VirtualHost	Nov, 21 2011	17.105.211.123	10.86	27.70	73.95	0	17.84	54.36	99.91	802
DS_HOUDINI_107	Nov, 21 2011	192.168.150.55	0.56	2.32	64.82	9.76 GB	99.41	99.46	99.48	899
DS_HOUDINI_001	Nov, 21 2011	192.168.150.153	0.15	4.99	57.76	1.97 GB	52.80	90.39	97.63	427
DS_IONWEB107	Nov, 21 2011	192.168.150.83	0.42	2.06	6.25	1.97 GB	57.00	57.70	58.47	80
DS_IONWEB	Nov, 21 2011	192.168.150.83	0.42	2.06	6.25	1.97 GB	57.00	57.71	58.48	80
DM_HOUDINI	Nov, 21 2011	192.168.150.55	0.57	2.40	64.87	9.76 GB	99.41	99.46	99.48	903
DS_HOUDINI	Nov, 21 2011	192.168.150.55	0.56	2.41	64.82	9.76 GB	99.40	99.46	99.48	904

Completed Downloaded 8 updates | Reset

Customer Information

This section contains information about accessing documentation online and how to contact Technical Support.

Accessing Documents Online

To access documents using a Web browser:

1. Go to the ION Trading® Web site at:

   http://www.iontrading.com/
2. Enter the ION Tracker Web site by logging in to the Client section.
3. Select **Docs & Packages** in the left hand frame.
4. Select one of the following:

   Tree Browser
   Search
   Configuration
   Data Reference

Providing Feedback about Documentation

If you have *comments or suggestions* about ION documentation, send an e-mail to techdocs@iontrading.com.

Contacting Technical Support

If you have a *problem*, you can contact ION Technical Support in one of the following ways:

- Telephone
- Web
- E-mail

Telephone Support

The service hours for all our clients in Europe and the U.S.A. are from 7:00 am to 6:00 pm (local time), Monday to Friday, and from 9.00 am to 5.00 pm in Japan.

EUROPE	7.00 am to 6.00 pm	+ 44 207 398 0222
JAPAN	9.00 am to 5.00 pm	+ 81 3 5404 8488
U.S.A.	7.00 am to 6.00 pm (EST)	+ 1 212 906 0050

Should you be unable to contact ION on any of the numbers above, you can also use the following number for emergencies:

+ 39 050 220 3800

Web Support

You can access the ION Tracker Web site at the following Web site:

http://www.iontrading.com/

The following information and services are available on this site:

- **News**
  The latest announcements on market changes and ION products.
- **Documentation**
  All the guides and release notes available in PDF format.
- **New releases**
  Each new version of a component will be available for download in this section.
- **Development plans and enhancements**
  Current development plans and tracking system of the enhancement requests submitted to ION.

E-mail Support

You can send e-mail to Technical Support at the following address:

support@iontrading.com

When you send e-mail, please provide the following information to accelerate the process:

- What platform is affected
- What component is involved
- The version of the component
- Detailed description of the problem
- Instrument codes involved
- Operators involved
- Detail of any error messages

Please provide logs whenever possible.

To allow us to gather as many elements as possible from the aftermath of crashes, customers are advised to enable the following options on machines running Microsoft Windows:

(Dr Watson configuration)

- Dump Symbol Table
- Dump All Thread Contents
- Create Crash Dump File

Remember to send all heavy log files or attachments to the following address:

bigfiles@iontrading.com.

Once you have logged an issue, you will be assigned a reference number. Please ensure you quote this number in all related communications.

Please send all other communications to sales@iontrading.com.