HLT2 Histogram Data Flow

Histograms are produced inside the HLT2 tasks running on the farm. These are added up by the HLT2 monitoring infrastructure and saved to saveset files (ROOT format) by the Hlt2Saver task that runs on machine hlt02.

The histograms are saved to separate files depending on which run the events that fill histograms belong to. Whenever a saveset file is written, the Hlt2Saver tasks publishes a message containing it's name and the path to the saved file.

Analysis tasks can subscribe to receive these messages, and a specific function is called whenever a message is received to allow processing of the histograms contained in it.

Any histograms produced by an analysis task can be published, at which point they are written to their own saveset. At that point, a message is published announcing the availability of the new saveset created by the analysis task, such that it can be analysed further.

To avoid having to know the precise machine where an application runs and ports to connect to to subscribe to messages about savesets it publishes, a central tasks runs that all tasks register to, and which can be queried to obtain sufficient information to connect to another task. This application is called the registrar. To see which tasks are currently registered, run the following command somewhere in the online environment, e.g. on plus:

```
$> lb-run Online/latest registrar_request.exe
```

Development Environment

To simplify development and merge requests, it is recommended to use a full checkout of the Online project as your development environment, and to work on a branch. Before setting this up, make sure gitlab access using ssh keys has been setup and your key is available on the machine where you plan to develop.

When working on a machine in the online environment, it is useful to add the following lines to a login script such as your .bashrc file:

```
$> export http_proxy=netgw01.lbdaq.cern.ch:8080
$> export https_proxy=netgw01.lbdaq.cern.ch:8080
```

To setup a development environment based on a checkout of the entire project, the https://gitlab.cern.ch/lhcb-HLT/trigger-dev[trigger-dev]] tools can be used. Follow the instructions in its README.md, setting the projects to build to Online.

This setup is assumed to be in use on the rest of the page, which means that an environment is available, which can be obtained using:

```
Online/run bash --norc
```

If the full project setup is not used, `Online/run` can be replaced by either `lb-run Online/latest` if local modifications are not required, or `./run` for a lb-dev area.

Writing an Analysis Task

An analysis task consists of two part, the task itself and its configuration.

A task is implemented by creating a python class that inherits from the `Analyser` class found in the `Monitoring.Analyser` module, part of the Online project starting with release v6r7. The class need only

implement the `analyse(app_name, file_name)` function:

Analysis Code

```
from Monitoring.Analyser import Analyser

class ExampleAnalyser(Analyser):
     def analyse(self, app_name, filename):
         self.Info('%s published %s' % (app_name, filename))
```

To publish (ROOT) histograms, the `publishHistogram` method can be used:

```
from ROOT import TH1D

class ExampleAnalyser(Analyser):
    def __init__(self, name, **kwargs):
        super(ExampleAnalyser, self).__init__(name, **kwargs)
        self._histo = TH1D("n_ana", "# savesets analysed", 1, -0.5, 0.5)

   def analyse(self, app_name, filename):
        self.Info('%s published %s' % (app_name, filename))

        self._histo.Fill(0)

        self.publishHistogram(self._histo, add=False)
```

If the number of the run that a saveset file belongs to needs to be known, it can be obtained from the saveset filename:

```
run = 0
try:
    run = int(filename.split('-')[1])
except (IndexError, ValueError):
    pass
```

The `publishHistogram` method takes the `runNumber` as an optional argument, which causes the published histogram to be written to a saveset for that `runNumber`. If no `runNumber` is passed, the histogram will be written to a saveset file that belongs to the run currently running (in the LHCb partition).

Configuration

To configure an analysis task, the `MonitoringJob` configurable should be used:

```
from Configurables import MonitoringJob
mj = MonitoringJob()
mj.JobName = "ExampleAnalyser"
```

Plese note: the value of the `JobName` option **must not** be the same as the name of the instance of any instances of classes that inherit from Analyser.

Instantiating and Starting the Analyser

Since the analysis tasks use GaudiPython, the application manager needs to be instantiated before the analyser itself can be instantiated, followed by initializing and starting the ApplicationMgr.

```
from GaudiPython.Bindings import AppMgr
gaudi = AppMgr()
```

```
reg_con = mj.getProp('RegistrarConnection')
analyser = ExampleAnalyser('ExampleAnalyser',
                           Applications=['Moore2Saver'],
                           RegistrarConnection=reg_con,
                           HistogramDirectory=mj.JobName,
                           OutputLevel=mj.getProp("OutputLevel"),
                           RunInMain=False)


from Monitoring.MonitoringJob import initialize, start
initialize(gaudi)

gaudi, monSvc = start()
```

Testing an Analysis Task

When testing an analysis task, some addtional configuration options are required:

```
mj.SavePath = '/tmp/histograms'
mj.DimDNSNode = "localhost"
```

The simplest way to test an analysis task is to run it on a local machine (e.g. plus) using interactive python:

```
$> python -i test_analyser.py
```

Once the task is running, it should show up in the list printed by:

```
$> registrar_request.exe
```

Testing this way relies on the production HLT2 saver task to be running, and your analyser will work on the saveset files it publishes. If the production HLT2 saver is not running (HLT2 is stopped), or you want to test things on specific savesets, read the next section.

To view the histograms in the presenter, a somewhat more convoluted command is required, otherwise the presenter will not detect the analysis task:

```
UTGID=LHCb_ANY_EXAMPLEANALYSER_00 sh -c 'exec -a $UTGID python -i test_analyser.py'
```

Once the task is running, the presenter can be pointed to the same machine to view any histograms being published:

```
 $>
presenter -D localhost -M editor-online
```

Double-click on localhost in the box at the top-right of the presenter window marked "DIM Histogram Browser". EXAMPLEANALYSER should be in the list, click on the + in front of it to navigate through the histograms it is publishing.

To view one, set the tick-mark in front of the histogram you want to view, right-click it and select "Add marked to page".

Completely Independent Test Setup

To setup a completely independent test setup, two additional programs need to be run: a local registrar and something that publishes savesets. A registrar can be started on any machine using the script:

```
$> python -i $MONITORINGROOT/scripts/registrar.py
```

To configure the example analyser to connect to the local registrar, include the following lines:

```
import socket
mj.RegistrarConnection = 'tcp://%s:31360' % socket.gethostname()
```

To publish an existing set of HLT2 saveset files the following script will impersonate the production saver. By default it will connect to a registrar running on the same machine on the default port:

```
$> lb-run Online/latest python -i $MONITORINGROOT/scripts/fake_publisher.py
```

If another task than the Moore2Saver needs to be impersonated, edit the script to reflect the required publisher and set of saveset files.

To check if applications have properly registered to the local registrar, use:

```
$> registrar_request.exe -h $HOST
```

Examples

The Monitoring.RunCompletion and Monitoring.DiskMonitor modules contain working examples analysis tasks. They obtain information from DIM services and the run DB , which is explained in the section on other sources of information.

Running an Analysis Task in Production

A virtual machine named analysis01 has been created to host tasks that run in production.

To run a task in production, two additional scripts are required, one for systemctl and one that starts the script. Once these have been created, request the systemctl startup script to be added to analysis01 by creating a ticket for the sysadmins (send a mail to lbonsupp@cernNOSPAMPLEASE.ch), mentioning that you'd like to run a new task on analysis01 and attach the proposed systemctl script.

Assuming that the production version of the MonitoringOnline project is used and the task name is RunCompletion, the scripts could look like the following:

Startup script for the system:

```
#!/bin/bash
#
# RunCompletion     Run an the run completion analyser
#
# Enrico Bonaccorsi
# Last modified:  Roel Aaij
# chkconfig: 345 99 03
# description: Activates/Deactivates the run completion analyser
#              to start at boot time.
#
### BEGIN INIT INFO
# Provides: RunCompletion
# Required-Start: AnalysisLogger
# Default-Start: 3 4 5
# Default-Stop: 0 1 2 6
### END INIT INFO
PATH=$PATH:/opt/FMC/bin/:/opt/FMC/bin/

NODE=analysis01
# SERVICENAME MUST BE THE SAME AS UTGID
NAME=RunCompletion
```

```
SERVICENAME="${NAME}_0"
LOGFIFO="/var/run/fmc/logAnalysisLogger.fifo"
LOCKFILE="/var/lock/subsys/${NAME}"
USER=online
GROUP=onliners
SCRIPT=/group/hlt/monitoring/MONITORINGONLINE/MONITORINGONLINE_RELEASE/Monitoring/Hlt2Monitoring/

start() {
        logger "Starting $SERVICENAME"
        pcAdd -m $NODE -u $SERVICENAME -n $USER -g $GROUP -O $LOGFIFO -E $LOGFIFO $SCRIPT
        touch $LOCKFILE
        logger "$SERVICENAME Started"
}

stop() {
        logger  "Stopping $SERVICENAME"
        pcRm $SERVICENAME
        rm -f $LOCKFILE
        logger "$SERVICENAME Stopped"
}

status() {
        pcLs
}

### main logic ###
```

Script executed by and referenced in the startup script:

```
#!/bin/bash
# ================================================================================
#
#  Script to start the run completion monitor
#
#  Author   R. Aaij
#  Version: 1.0
#  Date:    15/06/2018
#
# ================================================================================
#
export CMTCONFIG=x86_64-centos7-gcc62-dbg

# environment
. /group/hlt/monitoring/MONITORINGONLINE/MONITORINGONLINE_RELEASE/setup.$CMTCONFIG.vars

export UTGID
export LOGFIFO=/tmp/logAnalysisLogger.fifo
export PARTITION="LHCb"
export PARTITION_NAME="LHCb"
export DIM_DNS_NODE=mona08
export PYTHONPATH=/group/online/dataflow/options/${PARTITION}/RECONSTRUCTION:$PYTHONPATH

export PYTHONHOME=`python -c 'import sys; print sys.prefix'`

read COMMAND <<EOF
from Monitoring import RunCompletion;\
RunCompletion.run(DimDNSNode='mona08',\
                  OutputLevel=2)
EOF

echo "[DEBUG] RunCompletion Monitor on ${HOSTNAME}"
exec -a ${UTGID} python -c "${COMMAND}"
```

Other Sources of Information

Sometimes it is useful to obtain additional information from either DIM or the Run DB. Both of these information sources are easy to integrate.

The Run DB

To get information from the run DB, use the python wrapper around the json API:

```
from RunDBAPI import rundbapi
print rundbapi.get_run_info(run)
```

DIM Services

A lot of information is published through DIM services, often strings, but also integers, floats or doubles. All services are declared to a DIM DNS node and are visibile only on that node. Histograms are also published to the presenter through DIM, and to make them visible to the presenter used by the shifter, the DIM DNS node must be set to `mona08`.

How data from a DIM service can be obtained inside an analysis task depends on whether the service is available on `mona08`. To view services available on a given node, use the `did` command (the default DNS node is mona08):

```
$> lb-run Online/latest did -dns=dns_node
```

DIM services can be queried for their current value, or a callback can be registered that will be called whenever there is an update.

Different pieces of information that arrive asynchronously in analysis task, such as messages that a new saveset file has been written are handled using 0MQ. This usually result in an infinite loop that includes a call to poll for new messages. To integrate DIM services with this loop, some wrappers and adapters are available.

The Monitoring.RunCompletion and =Monitoring.DiskMonitor modules contain working examples of how this can be used.

Same DNS Node

If a service is available on the same DNS node that the analysis task is running on (typically `mona08`), the `DimMonitorSvc` can be used:

```
class ExampleAnalyser(Analyser):
    def initialize(self):
        sc = super(ExampleAnalyser, self).initialize()
        if not sc.isSuccess():
            return sc

        from GaudiPython.Bindings import gbl
        self._dimSvc = self.service(gbl.DimMonitorSvc, "DimMonitorSvc")

    def start(self):
        sc = super(ExampleAnalyser, self).start()
        if not sc.isSuccess():
            return sc
        self._runNumber = self._dimSvc.monitor('int')('RunInfo/LHCb/RunNumber')
        return sc

    def analyse(self, app, filename):
```

```
        self.Info('%s published %s, LHCb run %d' % (app_name, filename,
                                                    self._runNumber.get())
```

Other DNS Node

If a DIM service is only available on another DIM DNS node, a separate process that listens to the other node is required, which will forward information to a 0MQ socket. This is done by the `DimForwarder`. Multiprocessing is used to run it in a different process. I can be passed an optional pipe such that the main process can wait for it to be ready.

The `connection_path` passed to it will be used to open an ipc socket with connection path `ipc://your_path`, in the case below: `ipc:///tmp/RunCompletion`.

```
from multiprocessing import Process, Pipe
forward_pipe, start_pipe = Pipe()
run_svc = DimForwarder('/HLT/HLT1/Runs',
                       dim_dns_node="ecs03",
                       connection_path="/tmp/RunCompletion",
                       pipe=start_pipe)
forwarder = Process(target=run_svc)
forwarder.daemon = True
# fork must happen here, so start now
forwarder.start()

r = forward_pipe.poll(60)
if r:
    forward_pipe.recv()
else:
    print 'error'
```

To gracefully close the `DimForwarder`, it opens an ipc connection, and will close if "TERMINATE" is sent to it (the monSvc is returned by the call to start():

```
# Connect control connection
zmqSvc = monSvc.zmq()
control = zmqSvc.socket(zmq.PAIR)
control.connect(run_svc.control_connection())

control.send("TERMINATE")
if forwarder.is_alive():
    forwarder.join()
```

Synchronised Callback

If a callback from the DIM service is required to act on the information as soon as possible, race conditions with the call to `analyse` should be avoided. To achieve this, a handler can be registered, which will ensure things are synchronised. That will result in the function that is passed to it being called with the socket that is also passed as an argument whenever the socket receives a message. The socket has to be created, and can be connected to the connection provided by the forwarder:

```
from Monitoring.decorators import zmq
from Monitoring.DimMonitor import DimForwarder


run_svc = DimForwarder(...)


class ExampleAnalyser(Analyser):
    def update_info(self, socket):
        self.Debug('update_info')
        info = socket.recv_string()
        self.Info(info)
```

```
   def start(self):
        dim_socket = self.socket(zmq.SUB)
        dim_socket.setsockopt(zmq.LINGER, 0)
        dim_socket.setsockopt(zmq.SUBSCRIBE, "")
        dim_socket.connect(run_svc.service_connection())
        self.register_handler(dim_socket, zmq.POLLIN, self.update_info)
```

If the `DimMonitorSvc` is used, an additional argument can be passed when constructing the monitor object. Due to the nature of ownership in python, the socket passed to the dispactch function must be "kept" somehow, e.g. as a member variable.

This is a bit round-about as a result of the interaction between python and C++. If this functionality is used more, the possibility to pass a python function directly to the DimMonitor object could be added.

```
from Monitoring.decorators import zmq
from GaudiPython.Bindings import gbl

class ExampleAnalyser(Analyser):
    def initialize(self):
        sc = super(ExampleAnalyser, self).initialize()
        if not sc.isSuccess():
            return sc

        self._runNumber = None

        self._dimSvc = self.service(gbl.DimMonitorSvc, "DimMonitorSvc")

    def run_number(self, socket):
        r = socket.recv('int')
        self.Info("LHCb run number is now %d" % r)

    def start(self):
        sc = super(ExampleAnalyser, self).start()
        if not sc.isSuccess():
            return sc

        self._socket = self.socket(zmq.PAIR)
        self._socket.setsockopt(zmq.LINGER, 0)
        self._socket.bind("inproc://dispatch")
        fun = gbl.DimHelper('int').dispatcher(self.zmq(),
                                              self._socket)
        self._runNumber = self._dimSvc.monitor('int')('RunInfo/LHCb/RunNumber', fun)

        dim_socket = self.socket(zmq.PAIR)
        dim_socket.setsockopt(zmq.LINGER, 0)
        dim_socket.connect("inproc://dispatch")
        self.register_handler(dim_socket, zmq.POLLIN, self.run_number)

        return sc

    def analyse(self, app, filename):
        if self._runNumber is None:
            return
        self.Info('%s published %s, LHCb run %d' % (app_name, filename,
                                                    self._runNumber)
```

A Note on Normalization

At the moment, unfortunately, the histograms in the HLT2 saveset files cannot be trusted to all contain the same fraction of the data processed. This should be the case, but a fraction (about 50%) of the data is getting lost in transmission.

In addition, this fraction is not the same for different histograms, so relative fractions can also not be trusted. This issue is under investigation. If you would like to help in resolving it, please contact RoelAaij and/or the HLT group.

-- RoelAaij - 2017-09-18

This topic: LHCb > Hlt2AnalysisTasks
Topic revision: r10 - 2018-06-15 - RoelAaij