User's Guide

September 2000

Sergei Kolos

Note Number: 157
Reference: http://atddoc.cern.ch/Atlas/Notes/157/mon-ug.html

Online Monitoring

ATLAS DAQ

Version 1.1

This document has been prepared with Release 5 of the Adobe FrameMaker® Technical Publishing System using the User's Guide template prepared by Mario Ruggier of the Information and Programming Techniques Group at CERN. Only widely available fonts have been used, with the principal ones being:

Running text:	Palatino 10.5 pt on 13.5 pt line spacing
Chapter numbers and titles:	AvantGarde DemiBold 36 and 24 pt
Section headings	AvantGarde DemiBold 20 pt
Subsection and subsubsection headings:	Helvetica Bold 12 and 10 pt
Captions:	Helvetica 9 pt
Listings:	Courier Bold 9 pt

Preface

This document is the user's guide for the online monitoring of the ATLAS Online Software system. The online monitoring system is responsible for the event transportation from event samplers providing event fragment sources up to the users' monitoring tasks. This document gives an overview of the event distribution sub-system of the online monitoring and describes the event sampling and monitoring tasks APIs provided by this sub-system. Instructions for the event sampling and monitoring tasks development are included.

This document should be read by anyone going either to implement an event sampler application that is responsible for supplying events to the event distribution sub-system or to develop a monitoring tasks that reads event from the event distribution.

This document has been prepared by Sergei Kolos (Serguei.Kolos@cern.ch) based on the current implementation developed within the context of the On-line sub-system of the ATLAS TDAQ project.

Outline

Contents

Chapter 1

Introduction

This chapter gives a general overview of the online monitoring system describing it's purpose, structure and elements. It should be read by anyone who wants a general introduction to the component and by developers before they proceed to the later chapters.

1.1 Overview of the Online Monitoring system

The aim of the online monitoring system is to provide online samples of events to users' monitoring tasks within the DAQ-1 project. The online monitoring satisfies the requirements outlined in the user requirements document [1].

1.1.1 Online Monitoring Architecture

Due to the size and complexity of the DAQ, the online monitoring system will consist of many programs executing on several distributed computers connected through a network. Such a distributed system reflects the structure of the DAQ itself and will be implemented as a set of entities called event samplers, each with responsibility for one crate of the DAQ system. Each event sampler is responsible for sampling event data flowing through the DAQ system and transportation of these events to the event distribution sub-system (see Figure 1.1).

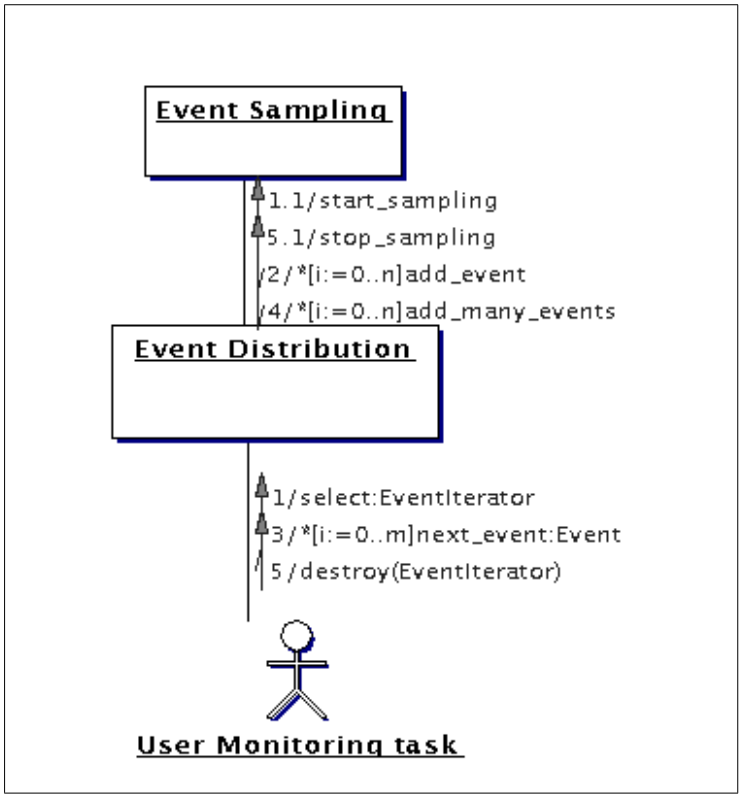

Figure 1.1 Online Monitoring architecture (UML Collaboration diagram)

The user monitoring task can request event fragments or full events with particular characteristics from the event distribution sub-system using it's public API that will be described later in this document.

The event distribution sub-system has a scalable architecture in order to be able to provide the reasonable event transportation performance independently of the size of the DAQ system itself and a number of monitoring task working concurrently. For the detailed information see [2].

1.1.2 Monitoring IDL description

Both the event sampling and the monitoring task interfaces provided by the event distribution sub-system are described in Object Management Group Interface Definition Language (OMG IDL)[3]. The OMG IDL is a part of the CORBA standard[4]. The interfaces described in OMG IDL can be implemented in different programming languages using different CORBA implementations. The complete OMG IDL interfaces for monitoring can be found in [2].

The public monitoring C++ API is built on top of the C++ classes generated by the IDL to C++ compiler from the IDL interfaces. For the moment, the Inter Language Unification (ILU) IDL compiler is used. The ILU system[5] is the CORBA implementation that has been used for a last few years by the Online software group. Although it's possible, it's not recommended to use the generated interface directly because it depends on the CORBA implementation specifics. The user's applications based on the generated interfaces will most probably require modifications if another CORBA implementation is used instead.

The monitoring API provide another level above the generated interface and hides completely the ILU peculiarities. The possible future swap of ILU with another ORB implementation does not affect the online monitoring API.

1.2 Online Monitoring information in the Information Service

The event distribution sub-system puts it's current statistical information to the Information Service (IS)[6]. The information is published when a new event buffer is created or a new monitoring task connects to the event distribution sub-system. This information is updated periodically with the time interval specified at the sub-system initialisation time (see Section 5.2).

By default, the event distribution component uses a dedicated IS server, named Monitoring, to store the information. This name can be redefined at the event distribution initialisation time (see Section 5.2).

There are two IS information types defined by the monitoring implementation. They are called ***Monitoring::BufferInfo*** and ***Monitoring::TaskInfo***. The ***Monitoring::BufferInfo*** holds the statistic about the event buffer used by the event distribution sub-system and contains the following information:

- Sampling Address is a structure of three character strings: detector id, crate id and module id. These values indicate which part of the DAQ system the events that are held by this buffer came from.
- Selection Criteria is a structure of four long integer values: trigger type, detector type, trigger state and event status word. These values show the characteristics of the events that are held by this buffer.
- Sampling mode that shows either events which are held by this buffer have been sampled on statistical basis or they are all the events of the dedicated type (defined by the Selection Criteria) available on the dedicated part of the DAQ system (defined by the Sampling Address).
- Size of the event buffer
- Number of events currently in the event buffer
- Total number of events that have been added to this buffer by event samplers
- Total number of events that have been accessed by monitoring tasks from this buffer
- Number of a monitoring tasks which are currently receiving events from this buffer
- Total number of monitoring tasks have ever received events from this buffer.

Another IS type that is called ***Monitoring::TaskInfo*** describes the statistics for the specific monitoring task and contains the following information:

- Sampling Address is a structure of three character strings: detector id, crate id and module id. These values indicate which part of the DAQ system the events came from.
- Selection Criteria is a structure of four long integer values: trigger type, detector type, trigger state and event status word. These values show the characteristics of the events which are requested by this monitoring task.
- Sampling mode that shows either events which are taken by this monitoring task have been sampled on a statistical basis or they are all the events of the

dedicated type (defined by the Selection Criteria) available on the dedicated part of the DAQ system (defined by the Sampling Address).

- Monitoring task presence, i.e. the monitoring task is now active or it has been stopped already.
- Number of events have been taken by this monitoring task.

For more details of how to access the online monitoring information in the IS see Chapter 4.

1.3 How the Online Monitoring affects the transported event data

This is very short but important chapter. The Online Monitoring implements automatic bytes swapping while transporting events between event samplers and monitoring tasks.The implementation is based on the assumption that the byte order for the event fragment is the same as the byte order for the machine on which the respective Event Sampler application is running. Based on this assumption the Monitoring Task API performs the byte swapping if the byte order for the machine on which the user Monitoring Task is running is different. This procedure is transparent for a user and is done by both Java and C++ APIs.

Chapter 2

Event Sampler development

This chapter describes how to develop an event sampler application that will be capable of sending the sampled events to the event distribution sub-system.

2.1 Introduction

The event sampler application is responsible for the communication with the data flow sub-system. It's implementation is specific for the different sub-detectors and DAQ crate types (e.g. ROD, ROC, SFC). The monitoring package provides only a skeleton class that defines an interface to the event distribution sub-system. A user wishing to carry out event sampling on his hardware must overload the methods in a "User" class which inherits from the ***Monitoring::EventSampler*** class. The following section describes the virtual methods of the ***Monitoring::EventSampler*** class and give an example of how they must be implemented.

2.2 Event Sampler Skeleton

This section describes how to overload and provide implementation for the virtual methods of the ***Monitoring::EventSampler*** class. The user's class that inherits from it is called ***MyEventSampler***. Listing 2.1 shows how to declare it.

Listing 2.1 My Event Sampler class declaration

```
 1:  class MyEventSampler: public Monitoring::EventSampler
 2:  {
 3:    public:
 4:          MyEventSampler( const IPCPartition & ,
 5:                          const char * detector_id,
 6:                          const char * crate_id  );
 7:
 8:          virtual Monitoring::Status   startSampling (
 9:                         const Monitoring::SamplingAddress & ,
10:                         const Monitoring::SelectionCriteria & ,
11:                         const MonitoringSampleAll & ,
12:                         Monitoring::EventAccumulator * );
13:
14:          virtual Monitoring::Status   stopSampling  (
15:                         const Monitoring::SamplingAddress & ,
16:                         const Monitoring::SelectionCriteria & );
17:
18:          virtual void   destroySampler ( );
19:  };
```

2.2.1 Event Sampler Constructor

According to the current monitoring system design it is necessary to have one Event Sampler running per DAQ crate. Therefore it must be possible to tell to the Event Sampler which crate it has to run on. This is done via the **detector_id** and **crate_id** parameters of the ***Monitoring::EventSampler*** class. Another parameter to the ***Monitoring::EventSampler*** class is the partition to which it belongs. For more information about partitions see [7]. Listing 2.2 shows how to implement a constructor for the customised event sampler class.

Listing 2.2 My Event Sampler constructor

```
1:  MyEventSampler::MyEventSampler(  const IPCPartition & p,
2:                                   const char * detector_id,
3:                                   const char * crate_id )
4:    :  Monitoring::EventSampler ( p, detector_id, crate_id )
5:  {
6:       ; // put here your sampler specific initialisation code
7:  }
```

2.2.2 Start Sampling request

The ***Monitoring::EventSampler::startSampling*** virtual method is called by the event distribution sub-system to request the Event Sampler to start collecting events which satisfy the ***criteria*** selection criteria and are taken from the DAQ part

identified by the ***address*** sampling address. Listing 2.3 shows the possible implementation for the ***startSampling*** method.

Listing 2.3 start Sampling method implementation

```
 1:  Monitoring::Status
 2:  MyEventSampler::startSampling (
 3:                    const Monitoring::SamplingAddress & address,
 4:                    const Monitoring::SelectionCriteria & criteria,
 5:                    const Monitoring::SampleAll & sample_all,
 6:                    Monitoring::EventAccumulator * accumulator )
 7:  {
 8:
 9:  // Check address: if it is invalid return Monitoring::BadAddress
10:
11:    if ( address.sa_detector.empty() )
12:    {
13:        return Monitoring::BadAddress;
14:    }
15:
16:  // Check the selection criteria: if it is invalid return
17:  // Monitoring::BadCriteria
18:
19:    if ( criteria.sc_detector_type < -1 )
20:    {
21:        return Monitoring::BadCriteria;
22:    }
23:
24:  // This global variable is used to stop the thread
25:  // when the stopSampling method is called
26:  // It will work for samplers with one sampling thread only !!!
27:  // User is responsable for implementation of more sophisticated
28:  // thread management technique
29:    Stop = 0;
30:
31:  // Create new thread and pass the Accumulator object to it
32:  // This thread will add events to the distribution sub-system
33:  // In this example the ILU procedure is used to start thread
34:  // Any other thread creation procedure can be used instead
35:
36:    ILU_ERRS((no_memory, no_resources,internal)) err;
37:    ilu_OSForkNewThread( EventSamplingThread, accumulator , &err );
38:
39:    return Monitoring::Success;
40:  }
```

The ***sample_all*** parameter defines either it is necessary to gather all the events flowing through the specific part of the DAQ system or it is enough to take some samples of them. The ***accumulator*** is the last parameter that represents a reference to the monitoring distribution event buffer. This reference must be stored somewhere in order to be used later for sending events to the event distribution sub-system. In Listing 2.3 this parameter is passed to the newly created programming thread (line 37). This thread is responsible for the communication with the data flow sub-system of DAQ. It must collect events according to the criteria that has been supplied by the ***startSampling*** method and feeds the event distribution sub-system with them using the ***Monitoring::EventAccumulator*** object reference. See Listing 2.5 for the example of the sampling thread implementation.

2.2.3 Stop Sampling request

The ***Monitoring::EventSampler::stopSampling*** virtual method is called by the event distribution sub-system in order to ask the Event Sampler to stop collecting events.

In response to this request the event sampler must terminate the event gathering that has been initiated by the ***startSampling*** request. The ***address*** and ***criteria*** parameters are used to identify the sampling thread that must be terminated. Listing 2.4 shows the possible implementation for the ***stopSampling*** method.

Listing 2.4 stop Sampling method implementation

```
 1:  Monitoring::Status
 2:  MyEventSampler::stopSampling (
 3:                const Monitoring::SamplingAddress & address,
 4:                const Monitoring::SelectionCriteria & criteria )
 5:  {
 6:
 7:  / Check address: if it is invalid return Monitoring::BadAddress
 8:
 9:    if ( address.sa_detector.empty() )
10:    {
11:        return Monitoring::BadAddress;
12:    }
13:
14:  // Check the selection criteria: if it is invalid return
15:  // Monitoring::BadCriteria
16:
17:    if ( criteria.sc_detector_type < -1 )
18:    {
19:        return Monitoring::BadCriteria;
20:    }
21:
22:  // Stop the thread that has been created
23:  // for these address and criteria
24:    Stop = 1;
25:
26:    return Monitoring::Success;
27:  }
```

2.2.4 Adding event to the event distribution sub-system

In order to add events to the monitoring distribution subsystem the Event Sampler application must use the ***addEvent*** and ***addManyEvents*** methods of the ***Monitoring::EventAccumulator*** class. As it is evident from the methods' names the first one feeds the distribution sub-system with one and the second with several events at once. The pointer to the ***Monitoring::EventAccumulator*** class that has

been passed as a parameter for the ***startSampling*** method must be used to call ***addEvent*** and ***addManyEvents*** methods. Listing 2.5 shows how this can be done.

Listing 2.5 Sampling thread implementation

```
 1:  // Event is an array of unsigned long integer values.
 2:  // We defined here the ExampleEvent as an array of
 3:  // four unsigned long integers
 4:
 1:  static unsigned long ExampleEvent[] =
 2:                           { 1L, 32767L, 65535L, 0xffff0000 };
 3:
 4:  // This function will be executed in a separate thread
 5:  // to simulate event sampling activity
 6:
 7:  static void    EventSamplingThread( void * param )
 8:  {
 9:
10:  // Cast the parameter to the accumulator's pointer
11:    Monitoring::EventAccumulator * accumulator =
12:                           (Monitoring::EventAccumulator*)param;
13:
14:    while ( Stop == 0 )
15:    {
16:        Monitoring::Status status = accumulator ->
17:            addEvent( ExampleEvent , 4 * sizeof(unsigned long) );
18:
19:        if ( status != Monitoring::Success )
20:        {
21:          cerr << "ERROR: Call to addEvent fails with code "
22:                 << (short)status << endl;
23:        }
24:
25:
26:   // Add 3 events at once
27:
28:       const unsigned int EventNumber = 3;
29:
30:       Monitoring::EventListlist( EventNumber );
31:
32:       for ( unsigned int i = 0; i < EventNumber; i++ )
33:       {
34:           list.add( ExampleEvent, 4 * sizeof(unsigned long) );
35:       }
36:
37:       status = accumulator -> addManyEvents( list );
38:
39:       if ( status != Monitoring::Success )
40:       {
41:           cerr << "ERROR: Call to addManyEvents fails with code "
42:                 << (short)status << endl;
43:       }
44:
45:    }
46:
47:   // Therad has been stopped
48:   // Destroy the Accumulator object
49:
50:    delete accumulator;
51:  }
52:
```

2.2.5 Destroying Event Sampler

There is one more virtual method for the ***Monitoring::EventSampler*** interface. It has the name ***destroySampler*** and is called when the event sampler must be destroyed. There is a default implementation for this method provided by the

Monitoring::EventSampler class. This implementation stops the CORBA main loop that has been started by the ***run*** method of the ***IPCServer*** class from which the ***Monitoring::EventSampler*** class inherits (see Section 2.3.1). If you did not call the ***run*** method or you need to perform some specific actions before exiting the sampler application you must overload the ***destroySampler*** method like the Listing 2.6 shows.

Listing 2.6 Destroy Sampler method implementation

```
 1:  void MyEventSampler::destroySampler( )
 2:  {
 3:
 4:  // Do whatever you need here
 5:  .......
 6:
 7:  // call the default implementation that stops the IPC main loop
 8:  // Do this only if IPCServer's run method has been called before
 9:
10:    Monitoring::EventSampler::destroySampler( );
11:  }
```

2.3 How to create and run the event sampler

This section describes how to create an instance of the customised event sampler and how to run it in a specific partition.

2.3.1 Instantiating and starting an event sampler

An example main program for starting an event sampler is shown in Listing 2.7. Since we assume that the event sampler is a multi-threaded application the ***IPCCore::init(TRUE)*** method must be called (line 23). This method creates a new thread that is responsible for the external requests processing.

The ***run*** method (line 33) is called to prevent the event sampler from exiting. This method is implemented as a conditional loop that is continuously calling the ***sleep(1)*** function until the ***stop*** method is called. The ***run*** and ***stop*** are the methods of the ***IPCServer*** class that is the base one for the ***Monitoring::EventSampler***.

You can use another system's main loop instead(e.g. X Window, TCL/Tk, or your own one). In this case you are responsible for the correct termination of the event sampler (see "Destroying Event Sampler").

Listing 2.7 Event sampler main function example

```
 1:  int main(int argc, const char * const * argv)
 2:  {
 3:
 4:    // parse command line parameters to get the partition_name,
 5:    // detector_id and crate_id
 6:    ...
 7:
 8:    // IMPORTANT: Init IPC core in Multi-threaded mode
 9:    IPCCore::init(TRUE);
10:
11:    // Define partition
12:    IPCPartition  p(partition_name);
13:
14:    // Instantiate the event sampler
15:    sampler = new MyEventSampler(p, detector_id, crate_id );
16:
17:    // Run the event sampler.
18:    // This method returns when
19:    // the IPCServer's stop method is called
20:    sampler -> run();
21:
22:    delete sampler;
23:
24:    return 0;
25:  }
```

Note
What has been explained above is valid for the multi-threaded applications only. If your event sampler is a single-threaded application the ***run*** method of the ***IPCServer*** class must be called. For a single-threaded application this method implements the main communication loop that is responsible for the external requests processing.

2.3.2 Compiling and linking an event sampler

Once all the necessary code to implement the required actions has been written to the user's event sampler class, the customised event sampler can be built by compiling and linking the code with the monitoring run-time library. The event sampler can then be tested in isolation or in conjunction with other event samplers using the test procedure and programs described in Chapter 5.

An event sampler needs to access header files for the monitoring and other Online software components (e.g. IS, IPC etc.). The monitoring package provides one library (shared on platforms that support shared libraries) needed for building an event sampler. This library is called ***libmon.so*** (***libmon.a*** on platforms that do not support shared libraries). In order to build an executable, a customised event sampler must be linked with the monitoring library as well as those provided by the following packages:

- IS
- IPC
- ILU
- Tools.h++
- CmdLine

For example, to compile and link an event sampler on Solaris the following makefile command can be used (see Listing 2.8).

Listing 2.8 Example makefile for the event sampler

```
 1:  #Online software root
 2:  RELEASE_DIR = /afs/cern.ch/atlas/project/tdaq/public/pro
 3:
 4:  PLATFORM_DIR = sparc-sun-solaris2.5.1/CC-4.2
 5:
 6:  INCLUDES = -I. -I${RELEASE_DIR}/include \
 7:             -I${RELEASE_DIR}/${PLATFORM_DIR}/include
 8:
 9:  LIBDIR = -L${RELEASE_DIR}/${PLATFORM_DIR}/lib
10:
11:  LIBS = -lmon -lcmdline -lis -lipc -lilu-c++ -lilu -lrwtool
     -lsocket -lthread
12:
13:  all: monitoring_sampler
14:
15:  monitoring_sampler : event_sampler_main.o event_sample_impl.o
16:       CC -o monitoring_sampler event_sampler_main.o \
17:             event_sample_impl.o -L. ${LIBDIR} ${LIBS}
18:
19:  %.o : %.cc
20:       CC ${CFLAGS} -c ${INCLUDES} $< -o $@
21:
```

Here it is assumed that the ***event_sampler_main.cc*** file contains the event sampler initialisation code and the ***event_sample_impl.cc*** file implements the virtual methods of the ***Monitoring::EventSampler*** interface.

Chapter 3

Monitoring Task development

This chapter describes how to implement a monitoring task that is able to retrieve events from the event distribution sub-system.

3.1 Introduction

The monitoring task API is implemented as a set of C++ classes, defined in the run-time library. A user wishing to carry out event monitoring simply needs to call the necessary methods in the appropriate monitoring classes. The following section describes those classes and methods and gives an example of how they must be called.

3.2 Monitoring task Implementation

This section shows an example of how to perform the event monitoring using the event distribution public API.

3.2.1 Monitoring initialization

The monitoring system initialization must be performed by creating an instance of the ***Monitoring*** class. The only parameter to the ***Monitoring*** class constructor is the partition in which the event monitoring will be performed. For more information about partitions see [7]. Listing 3.1 shows how to initialize the monitoring system.

Listing 3.1 How to initialize monitoring

```
22:    // Create a partition class instance
23:
24:    IPCPartition  partition(partition_name);
25:
26:   // Create a monitoring class instance
27:   // using partititon object as parameter
28:
29:    Monitoring    monitoring(partititon);
30:
```

3.2.2 Selecting Event Range and Origin

Once the monitoring system has been initialized one must define from which part of the DAQ system the events must be taken from and identify certain event characteristics used to select events. In other words, using the monitoring system's terms and definitions, it is necessary to specify the events' Sampling Address and Selection Criteria[2]. Listing 3.2 shows how to pass the Sampling Address and

Selection Criteria to the event distribution and get back the Event Iterator's reference using the ***select*** method of the ***Monitoring*** class.

Listing 3.2 Selecting range and origin of the events to be monitored

```
 1:    // Define Sampling address and selection criteria
 2:
 3:    Monitoring::SamplingAddress address( detector, crate, module);
 4:    Monitoring::SelectionCriteria criteria( 0, 0, 0, -1 );
 5:
 6:    // Declare the Event iterator's reference
 7:
 8:    Monitoring::EventIterator * it;
 9:    Monitoring::Status status;
10:
11:    status = monitoring.select( address, criteria, FALSE, it );
12:
13:    switch ( status )
14:    {
15:        case Monitoring::BadCriteria:
16:             cerr << "ERROR:: Invalid selection criteria." << endl;
17:             return 2;
18:        case Monitoring::BadAddress:
19:             cerr << "ERROR:: Invalid sampling address." << endl;
20:             return 3;
21:        case Monitoring::CommunicationError:
22:             cerr << "ERROR:: Communication error." << endl;
23:             return 4;
24:        default:
25:             break;
26:    }
27:    // The Event iterator's reference is valid here
28:    // and can be used to access the events
29:    ...
```

3.2.3 Non-blocking request for the next event

Using the pointer to the ***Monitoring::EventIterator*** object one can ask for the events available at that moment. The ***tryNextEvent*** method takes the reference to the ***Monitoring::Event*** pointer as a parameter and if at least one event is available in the event distribution system this pointer will be set to point to the newly created ***Monitoring::Event*** object. In this case the status that is returned by the ***tryNextEvent*** method will be set to the ***Monitoring::Success***.

It there are no events available the ***tryNextEvent*** method does not wait for their appearance and returns immediately with the status set to the ***Monitoring::NoMoreEvents*** and the ***event*** pointer remains unchanged.

Another possible error is the communication one. If the communication error appears the ***event*** pointer is unchanged and the returned status is the

Monitoring::CommunicationError. Listing 3.3 shows how to use the ***tryNextEvent*** method for the non-blocking event request.

Listing 3.3 Non-blocking request for the next event

```
 1:  while ( 1 )
 2:  {
 3:       // Declare the event pointer
 4:
 5:       Monitoring::Event * event;
 6:
 7:       // try to retrive the event
 8:
 9:       Monitoring::Status status = it -> tryNextEvent( event );
10:
11:       // Check the status of the tryNextEvent invocation
12:
13:       switch ( status )
14:       {
15:            case Monitoring::NoMoreEvents:
16:                 cerr << "WARNING:: No events are available for the
     moment." << endl;
17:                 continue;
18:
19:            case Monitoring::CommunicationError:
20:                 cerr << "ERROR:: Communication error." << endl;
21:                 exit( 1 );
22:
23:            default:
24:                 break;
25:       }
26:
27:       // Get the event's size ( number of unsigne long words)
28:       unsigned long   size = event -> size( );
29:       // Get the event data (the array of unsugned long words)
30:       unsigned long * data = event -> data( );
31:
32:       // print out the event data
33:       cout << "Event size : " << size << endl;
34:       cout << "Event data : " ;
35:       for ( unsigned long i = 0; i < size; i++ )
36:             cout << data[i] << " ";
37:       cout << endl;
38:
39:       // remove event if it is not needed anymore
40:       delete event;
41:  }
```

3.2.4 Blocking request for the next event

With the pointer to the ***Monitoring::EventIterator*** object one can ask for the events using the blocking communication style. If there are no events available for the moment the ***nextEvent*** method blocks for a number of seconds specified by the second method's parameter. This method also takes the reference to the ***Monitoring::Event*** pointer as the first parameter and if at least one event becomes available in the event distribution sub-system within the time-out period this pointer will be set to point to the newly created ***Monitoring::Event*** object. In this case the ***nextEvent*** method returns the ***Monitoring::Susccess*** value. If the specified period of time has elapsed and no events are available the method returns the ***Monitoring::Timeout*** value and the ***Monitoring::Event***'s pointer remains unchanged.

The communication error can also happens for this method. If the communication error appears the ***Monitoring::Event***'s pointer remains unchanged and the method

returns the ***Monitoring::CommunicationError*** value. Listing 3.4 shows how to use the ***nextEvent*** method for the blocking event request.

Listing 3.4 Blocking request for the next event

```
 1:  while ( 1 )
 2:  {
 3:       // Declare the event pointer
 4:
 5:       Monitoring::Event * event;
 6:
 7:       // try to retrive the event blocking for 10 seconds if there
 8:       // are no events available immediatly
 9:
10:       Monitoring::Status status = it -> nextEvent( event , 10 );
11:
12:       // Check the status of the nextEvent invocation
13:
14:       switch ( status )
15:       {
16:            case Monitoring::Timeout:
17:                 cerr << "WARNING:: No events are available within
     10 seconds." << endl;
18:                 continue;
19:
20:            case Monitoring::CommunicationError:
21:                 cerr << "ERROR:: Communication error." << endl;
22:                 exit( 1 );
23:
24:            default:
25:                 break;
26:       }
27:
28:       // Get the event's size ( number of unsigned long words)
29:       unsigned long   size = event -> size( );
30:       // Get the event data (the array of unsigned long words)
31:       unsigned long * data = event -> data( );
32:
33:       // print out the event data
34:       cout << "Event size : " << size << endl;
35:       cout << "Event data : " ;
36:       for ( unsigned long i = 0; i < size; i++ )
37:             cout << data[i] << " ";
38:       cout << endl;
39:
40:       // remove event if it is not needed anymore
41:       delete event;
42:  }
```

3.2.5 Destroying an Event Iterator

When the event iterator is not used anymore it must be deleted. This allows the event distribution sub-system to stop sampling events which are not requested anymore. Listing 3.5 shows how it must be done.

Listing 3.5 Destroying the event iterator

```
1:    // Destroy iterator if there is no need for it
2:
3:    delete it;
4:
```

3.3 How to create and run the monitoring task

This section describes how to build a monitoring task and run it in a specific partition.

3.3.1 Compiling and linking an monitoring task

Once all the necessary code has been written for the user's monitoring task, the monitoring task can be built by compiling and linking the code with the monitoring run-time library. The monitoring task can then be tested in isolation or in conjunction with other monitoring tasks using the test procedure and programs described in Chapter 5.

A monitoring task needs to access header files for the monitoring and other Online software components (e.g. IS, IPC etc.). The monitoring package provides one library (shared on platforms that support shared libraries) needed for building a monitoring task. This library is called ***libmon.so*** (***libmon.a*** on platforms that do not support shared libraries). In order to build an executable, a monitoring task code should be compiled and linked with the monitoring library as well as those provided by the following packages:

- IS
- IPC
- ILU
- Tools.h++
- CmdLine

For example, to compile and link a monitoring task on Solaris the following makefile command can be used (see Listing 3.6).

Listing 3.6 Example makefile for the monitoring task

```
 1:  #Online software root
 2:  RELEASE_DIR = /afs/cern.ch/atlas/project/tdaq/public/pro
 3:
 4:  PLATFORM_DIR = sparc-sun-solaris2.5.1/CC-4.2
 5:
 6:  INCLUDES = -I. -I${RELEASE_DIR}/include \
 7:             -I${RELEASE_DIR}/${PLATFORM_DIR}/include
 8:
 9:  LIBDIR = -L${RELEASE_DIR}/${PLATFORM_DIR}/lib
10:
11:  LIBS = -lmon -lcmdline -lis -lipc -lilu-c++ -lilu -lrwtool
     -lsocket -lthread
12:
13:  all: monitoring_sampler
14:
15:  monitoring_task : monitoring_task.o
16:       CC -o  monitoring_task monitoring_task.o \
17:                 monitoring_task.o -L. ${LIBDIR} ${LIBS}
18:
19:  %.o : %.cc
20:       CC ${CFLAGS} -c ${INCLUDES} $< -o $@
21:
```

Here it is assumed that the ***monitoring_task.cc*** file contains the monitoring task code described in this chapter.

Chapter 4

Class Reference

The Class Reference describes all the classes and functions in the Monitoring package. The Reference is organised as an alphabetical listing of classes. The entry for each class begins with a synopsis that lists the header files associated with the class, followed by an illustration showing the class's usage example. The synopsis also shows a declaration and definition of a class object. Following the synopsis is a brief description of the class and a list of member functions.

4.1 Monitoring

Synopsis

```
#include <monitoring/monitoring.h>
Monitoring monitoring( IPCPartition( partition_name ) );
```

Description

This class is used to define the range as well as the origin of the events to be monitored. In addition it specifies a partition for which online event monitoring is performed. All the other monitoring public API classes are defined as nested for the Monitoring class.

Example

```
//  Initialise Monitoring for the "TEST" partition

  IPCPartition p("TEST");
  Monitoring mon(p);

//  Declare the sampling address and selection criteria

  Monitoring::SamplingAddress   address( "Detector1", "Crate3",
      "Module7" );
  Monitoring::SelectionCriteria criteria( 0, 0, 0, -1 );

//  Declare the event iterator

  Monitoring::EventIterator * it;
  Monitoring::Status status;

//  Try to initialize the event iterator

  status = mon.select( address, criteria, FALSE, it );

  switch ( status )
  {
        case Monitoring::BadCriteria:
                cerr << "ERROR:: Invalid selection criteria." << endl;
                return 2;
        case Monitoring::BadAddress:
                cerr << "ERROR:: Invalid sampling address." << endl;
                return 3;
        case Monitoring::CommunicationError:
                cerr << "ERROR:: Communication error." << endl;
                return 4;
        default:
                break;
  }
```

Enumeration

```
enum Status { Success, CommunicationError,  Timeout, BadAddress,
              BadCriteria, NoMoreEvents };
```

Value of this type is returned by most of the Monitoring package methods.

Public Typedef

```
typedef unsigned char SampleAll;
```

Defines a type for the ‘sample all’ parameter of some Monitoring methods.

Public Constructor

```
Monitoring( const IPCPartition & p );
```

Constructs a monitoring object. The parameter **p** defines the partition in which events will be monitored.

Public Member Function

```
Monitoring::Status
select ( const SamplingAddress & sa, const SelectionCriteria & sc,
         const SampleAll & mode, EventIterator *& iter );
```

Creates an instance of the ***EventIterator*** class and sets **iter** to point to it. **iter** can be used to access the events selected according to the **sa**, **sc** and **mode** parameters.

Returns **Monitoring**::**Success** if event iterator has been created. Returns **Monitoring**::**BadAddress** if **sa** is invalid or **Monitoring**::**BadCriteria** if **sc** is invalid. Returns **Monitoring::CommunicationError** if the Monitoring Factory application has not been started or died or network is not functioning (see Section 5.4).

4.2 Monitoring::BufferInfo

Base Classes
ISInfo

Synopsis

```
#include <monitoring/monitoring_isinfo.h>
Monitoring::BufferInfo bi;
```

Description

This class is used to get the monitoring system buffers information from the Information Service (IS)[6].

Example

```
// Create IS iterator for the Monitoring information

ISInfoIterator ii( p, "Monitoring", ".*" );

Monitoring::BufferInfo bi;

while ( ii() )
{

  if ( bi.type() == ii.type() )
  {
     // Get information value from IS

     ii.value( bi );

     cout << "  { Detector = " << bi.address_.sa_detector
          << ", Crate = " << bi.address_.sa_crate
          << ", Module = " << bi.address_.sa_module << "}"
          << endl
          << "  { Detector type = " << bi.criteria_.sc_detector_type
          << ", Trigger type = " << bi.criteria_.sc_trigger_type
          << ", Trigger state = " << bi.criteria_.sc_trigger_state
          << ", Status word = " << bi.criteria_.sc_status_word << " }"
          << endl
          << "  { Sample All = " << (short)bi.sampleAll_ << " }"
          << endl
          << "Buffer size :     " << bi.size_ << endl
          << "Occupancy :       " << bi.occupancy_ << endl
          << "Events received : " << bi.eventsReceived_ << endl
          << "Events accessed : " << bi.eventsAccessed_ << endl
          << "Active tasks :    " << bi.activeClients_ << endl
          << "Total tasks :     " << bi.totalClients_ << endl
          << endl;
  }
}
```

Public Attributes

`SamplingAddress address_;`

Sampling address for the events held by this buffer.

`SelectionCriteria criteria_;`

Selection criteria for the events held by this buffer.

`SampleAll sampleAll_;`

Does this buffer contain all the events satisfying the **address_** and **criteria_** or just a statistical subset of them.

`unsigned long size_;`

Maximum number of events in the buffer.

`unsigned long occupancy_;`

Number of events in the buffer.

`unsigned long eventsReceived_;`

Total number of events that have been stored in the buffer.

`unsigned long eventsAccessed_;`

Number of events that have been requested by monitoring tasks.

`unsigned long activeClients_;`

Number of active monitoring tasks requesting events from this buffer.

`unsigned long totalClients_;`

Total number of monitoring tasks that have requested events from this buffer.

Public Constructors

`BufferInfo( );`

Constructs the monitoring buffer information object. All the public attributes are undefined immediately after construction. It is necessary to call one of the Information Service (IS) methods to initialize the object's attributes with the values taken from the IS [6].

4.3 Monitoring::Event

Synopsis

```
#include <monitoring/monitoring.h>
Monitoring::Event * event;
```

Description

This class represents the monitoring event allowing access to the event's data. It does not have the public constructor, therefore it can't be instantiated by user. User monitoring task can obtain a pointer to the object of this class by means of the ***nextEvent*** or ***tryNextEvent*** methods of the ***Monitoring::EventIterator*** class.

Example

```
Monitoring::EventIterator * it;

  // Initialise Iterator's pointer (see the Monitoring class example)
...

while ( stop == 0 )
{
        Monitoring::Event * event;

        status = it -> nextEvent( event, 10 );

        if ( status != Monitoring::Success )
        {
             break;
        }

        unsigned long size = event -> size( );
        unsigned long * data = event -> data( );

        cout << "Event size : " << size << endl;
        cout << "Event data : " ;
        for ( unsigned long i = 0; i < size; i++ )
        {
                cout << data[i] << " ";
        }
        cout << endl;

        // delete the event if it is not needed anymore

        delete event;
}
```

Public Member Functions

```
unsigned long *
data ( );
```

Returns a pointer to the event data. This method automatically performs byte swapping if it is necessary.

```
unsigned long
size ( );
```

Returns the number of elements (unsigned long integers) in the array that represents the event data.

4.4 Monitoring::EventAccumulator

Synopsis

```
#include <monitoring/monitoring.h>
Monitoring::EventAccumulator * accumulator;
```

Description

The pointer to this class is used by the Event Sampler implementations to push events to the event distribution sub-system. It does not have the public constructor, therefore it can't be instantiated by user. The event sampler implementation obtains the pointer to this class via the overloaded ***startSampling*** method of the ***Monitoring::EventSampler*** class.

Example

```
#include <my_event_sampler.h>

Monitoring::Status  MyEventSampler::startSampling (
                      const Monitoring::SamplingAddress & address,
                      const Monitoring::SelectionCriteria & criteria,
                      const Monitoring::SampleAll & sample_all,
                      Monitoring::EventAccumulator * accumulator )
{
  // Receive request to start sampling
  // Start a new thread and pass accumulator to it
  // NOTE: start_thread is a placeholder, the real thread
  // launching procedure must be used instead

    start_thread( thread_handler, accumulator );
}

  // Function that is executed in a separate thread
void    thread_handler( void * param )
{
  // Add events using accumulator
    Monitoring::EventAccumulator * accumulator =
                       (Monitoring::EventAccumulator*)param;

    Monitoring::Status status;

 // event_ptr is a pointer to the event data
 // event_size is the size of event data in bytes
    status = accumulator -> addEvent( event_ptr , event_size );
    if ( status != Monitoring::Success )
    {
        cerr << "ERROR: addEvent fails " << (short)status << endl;
    }

    Monitoring::EventList   list( EventNumber );

    for ( unsigned int i = 0; i < EventNumber; i++ )
    {
         list.add( event_ptr, event_size );
    }
```

```
    status = accumulator -> addManyEvents( list );

    if ( status != Monitoring::Success )
    {
        cerr << "ERROR: addManyEvents fails " << (short)status << endl;
    }
}
```

Public Member Functions

`Monitoring::Status`
`addEvent ( const void * data, unsigned long size );`

Add event to the event distribution sub-system. **data** is a pointer to the beginning of the memory block that holds the event data and **size** is the number of bytes for this data.

Returns **Monitoring::Success** if event has been successfully added. Returns **Monitoring::CommunicationError** if the Monitoring Factory application died or the network is not functioning (see Section 5.4).

`Monitoring::Status`
`addManyEvents ( EventList & list );`

Add a number of events held by **list** to the event distribution sub-system. Returns **Monitoring::Success** if the events have been successfully added. Returns **Monitoring::CommunicationError** if the Monitoring Factory applications died or the network is not functioning (see Section 5.4).

4.5 Monitoring::EventIterator

Synopsis

```
#include <monitoring/monitoring.h>
Monitoring::EventIterator * it;
```

Description

The pointer to this class is used by a monitoring task in order to get the events from the event distribution sub-system. It does not have a public constructor, therefore it can't be instantiated by user. A user must declare a pointer to the object of this class and call the ***select*** method of the ***Monitoring*** class in order to initialize this pointer.

Example

```
Monitoring::EventIterator * it;

  // Initialise Iterator's pointer (see the Monitoring class example)
...

while ( stop == 0 )
{
        Monitoring::Event * event;

        status = it -> nextEvent( event, 10 );

        switch ( status )
        {
                case Monitoring::Timeout:
                        cerr << "ERROR:: Timeout." << endl;
                        continue;
                case Monitoring::CommunicationError:
                        cerr << "ERROR:: Communication error." << endl;
                        exit( 1 );
                default:
                        break;
        }

        // process event and remove it when it is not needed anymore
        ...

        delete event;
}
```

Public Member Functions

```
Monitoring::Status
nextEvent ( Event *& event, unsigned long timeout )
```

Gets the next event from the event distribution sub-system and sets **event** to point to it. This method blocks for **timeout** seconds if no events are available.

Returns **Monitoring::Success** if event has been retrieved. Returns **Monitoring::CommunicationError** if the Monitoring Factory applications died or the network is not functioning (see Section 5.4). Returns **Monitoring::Timeout** if no new events become available within **timeout** period.

```
Monitoring::Status
tryNextEvent ( Event *& event )
```

Gets the next event from the event distribution sub-system and sets **event** to point to it. This method does not block and returns immediately if no events are available.

Returns **Monitoring::Success** if event has been retrieved. Returns **Monitoring::CommunicationError** if the Monitoring Factory applications died or the network is not functioning (see Section 5.4). Returns **Monitoring::NoMoreEvents** if no new events are available.

```
Monitoring::Status
reset ( );
```

Resets the iterator so that it references the first element of the event buffer. Returns **Monitoring::Success** if iterator has been reset. Returns **Monitoring::CommunicationError** if the Monitoring Factory applications died or the network is not functioning (see Section 5.4).

4.6 Monitoring::EventList

Synopsis

```
#include <monitoring/monitoring.h>
Monitoring::EventList list( event_number );
```

Description

This class is used by the Event Sampler implementations to transfer several events to the event distribution sub-system at once.

Example

```
    Monitoring::EventAccumulator * accumulator;

    // get a pointer to the accumulator (see EventAccumulator class)

    Monitoring::Status status;

    Monitoring::EventList   list( EventNumber );

 // event_ptr is a pointer to the event data
 // event_size is the size of event data in bytes

    for ( unsigned int i = 0; i < EventNumber; i++ )
    {
         list.add( event_ptr, event_size );
    }

    status = accumulator -> addManyEvents( list );
```

Public Constructor

```
EventList ( unsigned long n );
```

Constructs an empty event list. The constructor allocates memory for the **n** events. It will be possible to add more then **n** events to the list but it is inefficient due to the need for a reallocation procedure.

Public Member Functions

```
void
add ( const void * data, unsigned long size );
```

Add event to the list. **data** is a pointer to the beginning of the memory block that holds the event data and **size** is a number of bytes for this data.

4.7 Monitoring::EventSampler

Base Classes
IPCServer, IPCFreeableObject

Synopsis

```
#include <monitoring/monitoring.h>
class MyEventSampler : public Monitoring::EventSampler { ... };
```

Description

This class defines the Event Sampler interface. It declares two pure virtual methods that must be overloaded by an Event Sampler implementation.

Example

```
class MyEventSampler: public Monitoring::EventSampler
{
public:
        MyEventSampler( const IPCPartition & , const char * , const
      char *   );

        virtual Monitoring::Status startSampling (
                    const Monitoring::SamplingAddress & ,
                    const Monitoring::SelectionCriteria & ,
                    const MonitoringSampleAll & ,
                    Monitoring::EventAccumulator * );

        virtual Monitoring::Status stopSampling  (
                    const Monitoring::SamplingAddress & ,
                    const Monitoring::SelectionCriteria & );

        virtual void destroySampler ( );

};
```

Private Constructor

`EventSampler( const IPCPartition & p, const char * detector, const char * crate );`

Constructs an event sampler object. The parameter **p** defines the partition for which events will be sampled. **detector** and **crate** identify from which part of the DAQ system events will be taken by this sampler. The constructor tries to register the sampler within the partition **p**. If this attempt fails it prints an error message to the standard error stream and calls the ***exit(1)*** function.

Virtual Member Functions

```
virtual Monitoring::Status
startSampling ( const SamplingAddress & sa,
                const SelectionCriteria & sc,
                const SampleAll & mode,
                EventAccumulator *& acc) = 0;
```

This is a pure virtual method. It must be overloaded by the user defined class. This function causes the event sampler to start sampling events according to the **sa**, **sc** and **mode** parameters.

The user defined method must return **Monitoring::Success** if sampling has been successfully started. Otherwise, must return **Monitoring::BadAddress** if **sa** is invalid or **Monitoring::BadCriteria** if **sc** is invalid.

```
virtual Monitoring::Status
stopSampling ( const SamplingAddress & sa,
               const SelectionCriteria & sc ) = 0;
```

This is a pure virtual method. It must be overloaded by the user defined class. This function causes the event sampler to stop the sampling procedure that has been initiated earlier by ***startSampling*** request with the same **sa** and **sc** parameters.

The user defined method must return **Monitoring::Success** if sampling has been successfully stopped. Otherwise, must return **Monitoring::BadAddress** if **sa** is invalid or **Monitoring::BadCriteria** if **sc** is invalid.

```
virtual void
destroySampler ( );
```

This method can be overloaded by the user defined class that wants to perform some specific actions before Event Sampler destruction. There is a default implementation of this method that unregisters the sampler with the partition and stops the main loop that is responsible for processing incoming requests.

4.8 Monitoring::SamplingAddress

Synopsis

```
#include <monitoring/monitoring.h>
Monitoring::SamplingAddress address("Detector1", "Crate3", "Module7");
```

Description

This class is used to define the origin of the events to be monitored.

Example

```
//  Initialise Monitoring for the "TEST" partition

  IPCPartition p("TEST");
  Monitoring mon(p);

//  Define Sampling address and selection criteria

  Monitoring::SamplingAddress   address( "Detector1", "Crate3",
      "Module7" );
  Monitoring::SelectionCriteria criteria( 0, 0, 0, -1 );

//  Create Event iterator

  Monitoring::EventIterator * it;
  Monitoring::Status status;

  status = mon.select( address, criteria, FALSE, it );
```

Public Attributes

`string sa_detector;`

Specify id of the detector from which events must be sampled.

`string sa_crate;`

Specify id of the crate from which events must be sampled.

`string sa_module;`

Specify id of the module from which events must be sampled.

Public Constructors

`SamplingAddress( );`

Constructs the sampling address object and initializes all it's public attributes with **0**.

`SamplingAddress( const char * detector, const char * crate, const char * module );`

Constructs the sampling address object and initializes it's public attributes with the **detector**, **crate** and **module** values.

4.9 Monitoring::SelectionCriteria

Synopsis

```
#include <monitoring/monitoring.h>
Monitoring::SelectionCriteria criteria( 1, 2, 4, 0xffff );
```

Description

This class is used to define the peculiarities of the events to be monitored.

Example

```
//  Initialise Monitoring for the "TEST" partition

  IPCPartition p("TEST");
  Monitoring mon(p);

//  Define Sampling address and selection criteria

  Monitoring::SamplingAddress   address( "Detector1", "Crate3",
      "Module7" );
  Monitoring::SelectionCriteria criteria( 0, 0, 0, -1 );

//  Create Event iterator

  Monitoring::EventIterator * it;
  Monitoring::Status status;

  status = mon.select( address, criteria, FALSE, it );
```

Public Attributes

`long sc_detector_type;`

Sampled events must have this detector type.

`long sc_trigger_state;`

Sampled events must have this trigger state.

`long sc_trigger_type;`

Sampled events must have this trigger type.

`long sc_status_word;`

Sampled events must have this status word.

Public Constructors

```
SelectionCriteria( );
```

Constructs the selection criteria object and initializes all it's public attributes with **0**.

```
SelectionCriteria( long trigger_type, long detector_type,
                   long trigger_state, long status_word );
```

Constructs the selection criteria object and initializes it's public attributes with the **trigger_type**, **detector_type**, **trigger_state** and **status_word** values.

4.10 Monitoring::TaskInfo

Base Classes
ISInfo

Synopsis

```
#include <monitoring/monitoring_isinfo.h>
Monitoring::TaskInfo bi;
```

Description

This class is used to get the information about the user monitoring tasks which have been connected to the event distribution sub-system. This information is taken from the Information Service (IS)[6].

Example

```
// Create IS iterator for the Monitoring information

ISInfoIterator ii( p, "Monitoring", ".*" );

Monitoring::TaskInfo ti;

while ( ii() )
{

  if ( ti.type() == ii.type() )
  {
     // Get information value from IS

     ii.value( ti );

     cout << "  { Detector = " << ti.address_.sa_detector
          << ", Crate = " << ti.address_.sa_crate
          << ", Module = " << ti.address_.sa_module << "}"
          << endl
          << "  { Detector type = " << ti.criteria_.sc_detector_type
          << ", Trigger type = " << ti.criteria_.sc_trigger_type
          << ", Trigger state = " << ti.criteria_.sc_trigger_state
          << ", Status word = " << ti.criteria_.sc_status_word << " }"
          << endl
          << "  { Sample All = " << (short)ti.sampleAll_ << " }"
          << endl
          << "Events Received: " << ti.eventsReceiver_ << endl
          << "Is active :      " << ti.isActive_ << endl
          << endl;
  }
}
```

Public Attributes

`SamplingAddress address_;`

Sampling address for the events monitored by this task.

`SelectionCriteria criteria_;`

Selection criteria for the events monitored by this task.

`SampleAll sampleAll_;`

Does this task monitors all the events satisfying the **address_** and **criteria_** or just a statistical subset of them.

`unsigned long eventsReceived_;`

Total number of events have been received by this monitoring task.

`unsigned char is_Active_;`

Is this task performing monitoring now or it has been already stopped.

Public Constructors

`TaskInfo( );`

Constructs the monitoring task information object. All the public attributes are undefined immediately after construction. It is necessary to call one of the Information Service (IS) methods to initialize the object's attributes with the values taken from the IS [6].

Chapter 5

Building and Running Monitoring Applications

This chapter explains how to run the online monitoring system. These instructions can be used also for testing your own Event Samplers and Monitoring Tasks.

5.1 Event Sampler and Monitoring Task examples

There are two Monitoring example applications installed as a part of the Online software releases. The first one implements an Event Sampler simulator and another one implements a simple monitoring task. They will be explained in more detail in this chapter.

Note
Monitoring examples are included to Online software release and can be found in the $TDAQ_INST_PATH / com / monitoring / examples directory along with the makefile that can be tuned to be used with different operating system supported by the Online software.

5.1.1 Event Sampler example application

Event Sampler example shows how to accept the Start Sampling and Stop Sampling requests from the Monitoring Distribution subsystem and how to add events to it.

The Event Sampler example source code is distributed over three files:

– `event_sampler_impl.h` - declare the ***MyEventSampler*** class that inherits the ***Monitoring::EventSampler***

– `event_sampler_impl.cc` - implements the virtual methods of the ***Monitoring::EventSampler*** class

– `event_sampler_main.cc` - instantiates and runs the Event Sampler

5.1.2 Monitoring task example application

Monitoring task example shows how to request events from the Monitoring Distribution subsystem.

The Monitoring Task example source code is contained in the `monitoring_task.cc` file.

5.1.3 Makefile

The makefile is tuned by default to the Solaris 2.6 operating system and SunPro version 4.2 compiler. If you want to build examples for another operating system you have to modify the following makefile variables:

– PLATFORM_DIR - SRT target that defines OS / compiler combination (see makefile for the available targets)

– SOCKET_LIBS - socket library to be linked with (if necessary)

- THREAD_LIB - thread library to be linked with (if necessary)
- CC - compiler to be used
- RELEASE_DIR - make sure that RELEASE_DIR points to the root of Online software installation.

5.1.4 Compiling examples

You can compile and build the example applications by typing the following command in the monitoring/examples directory:

```
> make
```

5.2 Event Distribution implementation

The event distribution functionality are implemented by a single application called **monitoring_factory**. Below it is explained how to run this application.

Usage

```
monitoring_factory [-p partition-name] [-n is-server-name] [-l
event-limit] [-t time-interval]
```

Options and Arguments

`-p partition-name`	`partition to work in.`
`-n is-server-name`	`name of the IS server to store information (default is 'Monitoring').`
`-l event-limit`	`size of an event buffer (default is 100).`
`-t time-interval`	`time interval in seconds between the IS information updates (default is 60).`

5.3 Running Online Monitoring

This section describes how to start the online monitoring examples.

1. Setup your environment

Setup your environment so that you have access to the Online software via AFS (i.e. it is necessary to perform an AFS login or use the *klog* command to get a valid token) or from the CD-ROM. For details of how to access the software via AFS see:

```
http://atddoc.cern.ch/Atlas/DaqSoft/sde/srt_commands.html
```

For details of how to install the software from CD-ROM see:

```
http://atddoc.cern.ch/Atlas/CD-ROM/Welcome.html
```

2. Run partition

Start two IPC servers. One of them is used to run the general partition and another implements the named partition to be used by monitoring.

```
> ipc_server &
> ipc_server -p TEST &
```

3. Start the monitoring factory

Now run the monitoring factory application that provides the implementation for the event distribution sub-system:

```
> monitoring_factory -p TEST &
```

Monitoring factory will be started in the partition 'TEST'.

4. Run the Monitoring IS server

This step is optional. If you want to see the information that is published by the event distribution sub-system you must run the IS server. Run the IS server by typing the following command:

```
> is_server -p TEST -n Monitoring &
```

5. Run the Event Sampler

Run the Event Sampler example by giving the following command:

```
> monitoring_sampler -d Detector1 -c Crate3 -p TEST
```

The *-d and -c* switches mean the sampler will simulate event sampling from the crate 3 of the detector 1.

6. Run Monitoring Task

Run the Monitoring Task example by giving the following command:

```
> monitoring_task -d Detector1 -c Crate3 -p TEST
```

The *-d and -c* switches mean the monitoring task will request events from the crate 3 of the detector 1 using the default module and Selection Criteria hardcoded in this example.

7. Check the Monitoring IS information

You can check the monitoring IS information by executing the following command:

```
> monitoring_print_is -p TEST
```

5.4 Troubleshooting Tips

This chapter helps to resolve problems that may appear while trying to run the online monitoring. It describes the possible error messages and the actions that must be taken to resolve the problems.

5.4.1 Monitoring Factory errors

ERROR [MonitoringFactory::MonitoringFactory()] Cannot publish Monitoring Factory object in the partition default

This error indicates that the default partition server has not been started or has died or the network is not functioning. See Section 2 of Chapter 5.3 for how to start the default IPC server.

ERROR [MonitoringFactory::MonitoringFactory()] Cannot publish Monitoring Factory object in the partition PARTITION_NAME

This error indicates that the IPC server for the PARTITION_NAME partition is not running or the network is not functioning. See Section 2 of Chapter 5.3 for how to start the named IPC server.

5.4.2 Event Sampler errors

ERROR [Monitoring::EventSampler::EventSampler()] Cannot publish Monitoring Event Sampler object in the partition default

This error indicates that the default partition server has not been started or has died or the network is not functioning. See Section 2 of Chapter 5.3 for how to start the default IPC server.

ERROR [Monitoring::EventSampler::EventSampler()] Cannot publish Monitoring Event Sampler object in the partition PARTITION_NAME

This error indicates that the IPC server for the PARTITION_NAME partition is not running or the network is not functioning. See Section 2 of Chapter 5.3 for how to start the named IPC server.

5.4.3 Monitoring Task errors

ERROR::Communication error

Indicates that monitoring factory application is not running or network is not functioning. See Section 3 of Chapter 5.3 for how to start the monitoring factory.

ERROR::Invalid sampling address

Indicates that a specific sampler application is not running. See Section 5 of Chapter 5.3 for how to start the event sampler.

Bibliography

1 **User Requirements for the Online Monitoring of the ATLAS DAQ/EF Prototype -1**
M. Caprini, D. Francis, R. Jones, S. Kolos, J. Petersen, L.Tremblet , ATLAS DAQ/EF Prototype -1 project technical note 152,
http://atddoc.cern.ch/Atlas/Notes/152/Note152-1.html.

2 **Online Monitoring Design**
M. Caprini, R. Jones, S. Kolos, L.Tremblet, ATLAS DAQ/EF Prototype -1 project technical note 155, http://atddoc.cern.ch/Atlas/Notes/155/Note155-1.html.

3 **CORBA/IIOP 2.3.1 Specification, chapter 3-IDL Syntax and Semantics,**
http://cgi.omg.org/cgi-bin/doc?formal/99-07-07

4 **CORBA official Home Page,**
http://www.omg.org/corba/

5 **Inter-Language Unification,**
ftp://ftp.parc.xerox.com/pub/ilu/ilu.html

6 **Implementation of the Information Service: Users Guide,**
S.Kolos, ATLAS DAQ/EF Prototype -1 project technical note 37,
http://atddoc.cern.ch/Atlas/Notes/037/Note037-1.html.

7 **Inter Process Communication package,**
S.Kolos, ATLAS DAQ/EF Prototype -1 project technical note 75,
http://atddoc.cern.ch/Atlas/Notes/075/Note075-1.html.

Index