A reimplementation of the CORBA-based Event Monitoring System for the ATLAS LHC Experiment at CERN

Ingo Scholtes

Diploma thesis under supervision of Prof. Dr. Peter Sturm
Systemsoftware and Distributed Systems
University of Trier

November 10, 2005

\[\text{Ingo.Scholtes@cern.ch}\]
Abstract

The Atlas Event Monitoring system is a subsystem of the Atlas Online Software, providing a framework for sampling events from different stages of the data flow chain of the ATLAS detector and distributing it to so-called Monitoring Tasks run by physicists. After a short introduction to the ATLAS experiment, the data acquisition project and its different software components and the general purpose of Event Monitoring, this thesis discusses the previous implementation as well as different aspects of the reimplementation that has been done in the context of this thesis. After having done a theoretical analysis of the communication complexity of both implementations and the presentation of practical performance test results captured during the Large Scale Test period a conclusion will be made, showing why the reimplementation is more suitable for actual usage in the final experiment than the previous implementation.
I hereby want to thank my supervisor at CERN, Sergei Kolos for the support offered in the context of this thesis.
Erklärung zur Diplomarbeit


Datum

Unterschrift
Contents

1 Introduction
   1.1 ATLAS Experiment .............................................. 7
   1.2 ATLAS triggers .................................................. 8
   1.3 ATLAS Online Software ......................................... 10
      1.3.1 Infrastructure ............................................ 10
   1.4 ATLAS Event Monitoring ....................................... 11
      1.4.1 Introduction ............................................... 11
      1.4.2 Requirements ............................................. 13

2 CORBA and IPC .................................................. 15

3 Previous Implementation ......................................... 19
   3.1 Architectural Design .......................................... 19
   3.2 API usability .................................................. 19
   3.3 Drawbacks ...................................................... 20

4 Reimplementation ................................................ 21
   4.1 Architectural Design .......................................... 21
   4.2 API Usability .................................................. 23
      4.2.1 Push vs. Pull Model .................................... 24
      4.2.2 Genericity considerations and design pattern ......... 25
      4.2.3 System APIs ............................................. 27
   4.3 Implementation details ....................................... 36
      4.3.1 IDL declaration .......................................... 36
      4.3.2 Thread Safety Considerations ........................... 37
      4.3.3 Smart Pointer Memory Management ..................... 38
4.4 Scalability ............................... 38
4.5 Fault Tolerance .......................... 39
   4.5.1 Crash of an Event Sampler application .......................... 39
   4.5.2 Crash of a Monitoring Task application ....................... 39
   4.5.3 Crash of the Conductor ................................. 40
4.6 Congestion Control Scheme .......................... 41
4.7 Segmented Memory Support .......................... 42

5 Communication Complexity 45
   5.1 Definitions ............................. 45
   5.2 Previous Implementation ....................... 45
      5.2.1 Event Sampler .......................... 45
      5.2.2 Monitoring Task .......................... 46
      5.2.3 Distributor ............................ 47
   5.3 Reimplementation ............................ 48
      5.3.1 Monitoring Task .......................... 49
      5.3.2 Conductor ............................. 50

6 Performance Tests 51
   6.1 Large Scale Tests .......................... 51
      6.1.1 Network topology of Large Scale Testing Cluster ........... 51
      6.1.2 Large Scale Test Aim ........................... 51
      6.1.3 Large Scale Test Configurations ......................... 52
      6.1.4 Large Scale Test Approach and Execution ................. 52
      6.1.5 Large Scale Test Results .......................... 53
      6.1.6 Summary .................................. 55
      6.1.7 Future Tests .............................. 55
   6.2 Scalability Comparison .......................... 56
      6.2.1 Test Results ............................. 56
      6.2.2 Summary .................................. 57

7 Conclusion and Future Steps 59

A ATLAS Event Monitoring System Class Reference 61

B ATLAS Event Monitoring System Examples 139
Chapter 1

Introduction

1.1 ATLAS Experiment

The ATLAS detector is a particle physics experiment searching for new fundamental particles in head-on collisions of protons accelerated in the LHC, which is a 27 km circumference accelerator ring in the underground of the CERN Laboratory in Geneva, Switzerland. Using this accelerator it is possible to create an energy density, close to that of the Big Bang, revealing the fundamental building blocks of nature. After two first stages of acceleration in the PS and SPS, proton beams are injected to the LHC, where each of the two counterrotating beams is accelerated to an energy of up to 7 GeV, resulting in a collision energy of 14 GeV in the centre of mass. The general outline of the LHC experiment can be seen in figure 1.1. The mass of the proton will be totally annihilated in this collision and will according to Einstein’s famous formula \(E = mc^2\) be turned into energy, which then turns to mass again, producing various kinds of fundamental particles. These particles, which simplified can be seen as collision debris, will then be detected by the ATLAS detector, which is a huge cylinder of the size of a five storey building, surrounding the interaction point. Having a length of 42 m and a diameter of 22 m, the ATLAS detector consists of different layers, that can be compared to the different kinds of human senses. Whereas the innermost layer, the inner tracker measures the momentum of each crossing charged particle, the calorimeter measures the energy of each particle. The muon spectrometer, which is the outermost layer of the ATLAS detector, identifies and measures muons, which although they are 200 times heavier than electrons have virtually no interaction with mass. Another major component of the ATLAS detector is the magnet system, which basically bends the trajectory of charged particles for momentum measurement. Combining all these layers, the detector provides a full view of every collision event. Considering the fact that the detector provides roughly about 2 MB of data per collision event at a collision rate of 40 MHz, the ATLAS detector provides roughly 80 TB of raw collision event data per second. As one might imagine this can not be handled by any computing facility, so some form of selection has to be done, which events are worth being processed and which are not. This selection is done by so-called triggers which will be discussed in more detail in chapter 1.2.
1.2 ATLAS triggers

As mentioned in chapter 1.1 the data amount the ATLAS detector produces per second is far too large to be processed directly. Therefore, three stages of so-called triggers are used to reduce this amount of data so it can be transferred to mass storage. Considering, that the vast majority of collision events occurring in the interaction point are well known and understood, physicists are only interested in a very small fraction of these events. For the final experiment according to [1] it is estimated that only 10 to 100 per one billion events actually contain interesting data. The purpose of the different trigger levels therefore is to reduce the event rate significantly and effectively, filtering uninteresting events and letting pass only interesting events. Criteria for the quality of the trigger system are the number of well-known events passing the triggers, the fraction of potentially interesting events being dropped, the efficiency of the event rate reduction and the robustness of the system. Especially the front-end electronics of the ATLAS detector needs to be robust against the very high level of radiation that will be present in the ATLAS cavern when the actual experiment starts. Having an input rate of 40 MHz (i.e. a new event arrives every 25 ns) the first level trigger is implemented as a set of hardware processors specially built for this purpose and which take a maximum of 2.5 μs to decide, whether an event should pass or be rejected. In order to allow this processing time events are stored in pipelined buffers until a decision has been made [2]. According to [3] the ATLAS first level trigger drops 99.8125 % of the events, having an output rate of about 75 KHz. Considering an event size of roughly 2MB, this is still a data volume of 150 GB per second. The second level trigger, having an input event rate of 75 KHz (i.e. a new event from the first level trigger output arrives
roughly every 13.5 $\mu$s) again reduces the event rate by 96% to 3 KHz ([3], p. 51). Whereas the first two trigger levels operate upon event fragments coming from different detection areas of the ATLAS detector, the third stage's decision is for the first time based on the whole event data, as it is done after the process of event building in the EventFilter. This third decision stage which basically consists of a cluster of about 1500 dual-CPU machines now can take up to 1 second per event for its decision ([3], pp. 16f), reducing the event rate by 90% to 300 Hz. Having a data volume of about 600 MB per second, event data can now be transferred to mass storage devices using optical fibre connections, where they wait for analysis. All in all the whole trigger system offers a data amount reduction by 99.99925% or more than 5 orders of magnitude, reducing 80 TB per second to 600 MB. The overall design of the whole trigger system can be seen in figure 1.2.

Figure 1.2: The ATLAS Trigger System ([3], p. 6)
1.3 ATLAS Online Software

The ATLAS Online Software is a subsystem of the ATLAS Trigger-Daq(TDAQ) Project, providing a glue of different of its components. According to [4] its main purpose is the configuration, monitoring and controlling of the TDAQ system, having interfaces to the data flow, triggers, Read-Out Crates (ROC), Detector Control System (DCS), Event Builder (EB) and the processor farm. The Online Software is organized in different subsystems - some of them can be seen in figure 1.3.

Figure 1.3: Atlas Online Software Component Decomposition

1.3.1 Infrastructure

In order to simplify and unify the implementation of the different components of the ATLAS Online Software it provides its own middleware components for different purposes, which will be described in more detail in the following sections.

1.3.1.1 Interprocess Communication (IPC)

Based on CORBA, using omniORB 4, the IPC component provides the basis for all interprocess communication in the ATLAS Online Software components.


Implemented on top of CORBA COS Naming, it hides CORBA specific implementation details from the users and introduces the term Partition that can be seen as some kind of namespace for CORBA communication objects [14]. The ipc package along with some CORBA basics will be presented in more detail in chapter 2.

1.3.1.2 Information Service (IS)

The IS according to [4] provides a framework for publishing information in an IPC Partition, allowing communication objects to insert, update and remove information objects, as well as subscribing to information channels, receiving a callback on change of a piece of information. Information in IS are stored in a persistent manner.

1.3.1.3 Message Reporting System (MRS)

The MRS component provides a framework for message distribution inside an IPC Partition. Communication objects can send messages to a MRS server, as well as subscribe to a MRS server, receiving asynchronous callbacks on arrival of new messages. The main difference to IS is, that all information in the MRS server is transient. In the TDAQ system, MRS is mainly used for error reporting and exchange of information messages. According to [7] further important features of the MRS system include message definition, distribution, routing and filtering.

1.4 ATLAS Event Monitoring

1.4.1 Introduction

The ATLAS Event Monitoring System is a subsystem of the ATLAS Online Software, providing a software framework which allows users to write applications which sample collision event data from different stages of the data flow chain between ATLAS detector and mass storage and distribute event data efficiently to user-implemented event consumer applications. The term event whenever used in the context of this thesis does not refer to the asynchronous event mechanism provided by CORBA but to a collision event between two protons in the detector. When speaking of users in the scope of ATLAS Event Monitoring one generally means physicists who use the framework in order to develop applications which provide or consume event data. Event providing applications may then be deployed and attached to different well-defined points in the data flow chain. The framework allows event consuming applications to be implemented either in C++ or Java, event providing applications may only be implemented using C++. According to [5] the main purpose for sampling events from the data flow chain between detector and mass storage in the final ATLAS experiment will be the possibility to monitor data acquisition state and quality of physics data (e.g. collision event data) flowing through the system. As mentioned in [8] one example for the usage of the Event Monitoring System in the final experiment might
be the monitoring and confirmation of trigger decision quality (as described in chapter 1.2) or the identification of malfunctioning parts by the shift crew in the control room near the ATLAS pit.

The ATLAS Event Monitoring System is implemented on top of the IPC package described in 1.3.1.1. According to [5] the general architecture consists of the Event Monitoring Framework, Event Providers providing event data and Event Consumers receiving and processing these data. The Event Monitoring System provides two basic interfaces which may be utilized by the users in order to use the framework: The Event Provider needs to implement the Event Sampler interface in order to provide events to the Event Monitoring System, the Event Consumer may use the Event Iterator interface in order to retrieve events from a certain Event Provider. Throughout this thesis the Event Provider application will be called Event Sampler, the Event Consumer will be called Monitoring Task. Apart from the implementation of the Event Sampler interface, Event Sampler applications shall provide the code that actually samples events or event fragments matching a certain selection criteria requested by the Monitoring Task. Monitoring Tasks will also provide code that actually analyzes event data using for example standard CERN data analyzation frameworks like ROOT\(^1\).

\(^1\)see http://root.cern.ch/ for more details
1.4.2 Requirements

According to [9] the ATLAS Event Monitoring System shall allow user programs to sample event data flowing through the data flow chain including the ROD crates, while having a minimal and deterministic effect on the data flow performance. It shall enable users to request event fragments as well as full events from different stages of the data flow system, such as ROD, ROS or EventBuilder. It is important to observe, that a guarantee regarding the reception of the same event in different Monitoring Tasks is not required. The ATLAS Event Monitoring requirements [9] follow the guidelines proposed by [10] using the words "shall" for mandatory requirements, the word "should" for requirements that need a strong justification if not fulfilled and the word "may" for requirements that may not be fulfilled without justification at all. Using this terminology and citing from [9], in more detail the ATLAS Event Monitoring System

- shall provide event fragments to users Monitoring Tasks upon request
- shall provide full events to users Monitoring Tasks upon request
- shall provide a unified framework for sampling of events from all levels of the data flow chain
- shall allow running of multiple concurrent and independent Monitoring Tasks
- shall allow event fragments to be requested according to a subdetector identifier
- shall have a minimal and deterministic effect on the data flow performance
- shall limit the maximum number of event types allowed during a run
- shall allow the event types to be defined before the start of data taking
- should allow event fragments to be requested according to a single crate in a given subdetector
- should allow event fragments to be requested according to a single module in a given crate
- should allow event fragments to be requested according to a level 1 trigger type, a subdetector type, a level 2 filter decision, a status word in the event header and a combination of the afore mentioned
- should allow event fragments of a specific subdetector to be requested according to a subdetector dependent range
- should provide the possibility to request all events flowing through the DAQ system with a well defined effect on the DataFlow performance
- should be compatible with the main analysis package(s) used at CERN (ROOT, etc.)
Chapter 2

CORBA and IPC

CORBA is a standard for distributed object-oriented programming. Its vendor-independent architecture, standardized by the OMG - a consortium of several hundred computer industry companies - basically consists of clients, CORBA objects, servants and an object request broker (ORB), tied together by stubs and skeletons. A CORBA object is a programming entity used by clients, consisting of an interface, an object reference and a servant. A servant is the implementation of a CORBA object's interface. It is defined in the interface definition language (IDL), which is another central component of CORBA. Client stubs and object implementation (servant) skeletons are automatically generated from the IDL declaration of an object's interface by an IDL compiler. As there are several language bindings for IDL, clients and servants can be implemented using a wide variety of programming languages like C, C++ or Java. The ORB is responsible for handling object requests, localization of objects (whether they are remote or local), mapping servants to CORBA objects and performing requests and eventually returning data to the client. There may be one servant for one object, several servants for one object (one after the other over a period of time) or one servant for several objects. The mapping of CORBA objects to servants as well as its activation and deactivation is managed by the so-called object adaptor. The basic object adaptor (BOA) was the first object adaptor standard defined by the OMG, but it proved to be a bit too simplistic for usage in real life infrastructures, so many CORBA implementations added custom features in order to overcome the shortcomings. In order to keep code from being ORB-dependent because of these custom features added by ORB implementors, the OMG introduced a more sophisticated object adaptor standard, the so-called portable object adaptor (POA) ensuring interoperability of CORBA objects across different ORB implementations. In addition to many advanced features, the POA introduces threading policies. Servant objects can either be single-threaded, thus resulting in sequential requests or multi-threaded, thus resulting in concurrent requests requiring the servant code to be threadsafe. Every object instance has an unique identifier (its type being determined by the IDL interface implemented by this object), that can be used by the ORB in order to identify the exact object instance a client wants to access. In order
to allow remote object invocations, the IIOP (an IP-version of the General Inter ORB Protocol, GIOP) has been specified by the OMG, which is used for communication between ORBs running on different systems, allowing clients to access objects located on remote machines. Remote CORBA objects can be accessed via object references by a client just like local objects. While the client can not tell from the object reference if the servant is running on the local machine or remote, the ORB can and will route the request to the corresponding object (e.g. via the IIOP protocol). Figure 2.1 shows how in CORBA’s architecture clients, servants, stubs, skeletons, ORB, object adaptor and IDL work together. In this figure an example scenario with two servants for two objects A and B is considered. For the client, the servant for A is local, the servant for B is remote. Object requests will therefore be routed to another ORB via IIOP.

Figure 2.1: CORBA Architecture

Object references can be passed between applications by different means. One technique are so-called IOR-strings, which basically are Base64 codings of IP-Address, port and ORB-version of the hosting ORB along with the unique object identifier of the object instance. After an application has converted a CORBA object reference to an IOR string and this string has been passed from one application to another (e.g. via a distributed filesystem), the stringified object
reference can now be converted back to an object reference and be used to access the object.

As using IOR strings provides a simple yet not convenient way to pass CORBA object references between applications, there is another, more comfortable technique. The CORBA Naming Service called COS (Common Object Services) Naming defines a tree-like hierarchy of names and so-called naming-contexts in which names are unique. In order to use a COS Naming Service, all you have to know is the object reference of the COS Naming Service object. From now on, CORBA object references can be simply accessed via human readable names rather than passing around IOR strings. The problem of retrieving the initial object reference of the CORBA Naming Service is often referred to as bootstrapping. In a distributed system it might be solved by passing the root reference to the ORB on startup or by storing an IOR-stringified reference to it on a distributed filesystem at a well-known location.

The IPC component of the ATLAS Online Software is implemented on top of the omniORB ORB implementation, hiding CORBA specific details from the users of the IPC package, by defining a term called IPC-Partition. An IPC-Partition can be thought of like a CORBA namespace, in which all objects are separated from other partitions. When an IPC-server starts, it dumps its IOR object reference to a file defined in the environment variable $DAQ_INIT_IPC_REF$. All client application may now use the IOR object reference in this file in order to connect to the ipc server and resolve symbolic names of ipc objects. The IPC package defines an IDL interface $ipc::servant$ which IPC object interfaces need to inherit in order to be accessible via IPC. IPC is based on omniORBs portable object adaptor (POA). The threading policy to be used by an IPC object can be specified by inheriting the object implementation from $ipc::single_thread$ or $ipc::multi_thread$. 

Chapter 3

Previous Implementation

3.1 Architectural Design

In the previous implementation an application called Event Distributor plays a central role for event data distribution. When a user’s Monitoring Task starts monitoring event data, it will first send a SELECT message to the Event Distributor, returning an Event Iterator handle. The Event Distributor will then create a local buffer and invoke SELECTSAMPLER in the appropriate Event Sampler application. The user providing the sampling code will have to care about starting a thread and taking event data, that will be sent to the Event Distributor. The Event Distributor will then buffer these data, through which Monitoring Tasks may iterate by using the Event Iterator handle returned earlier. If another Monitoring Task starts monitoring the same selection criteria in the same Event Sampler, it will be silently attached to the buffer priorly created in the Event Distributor by receiving an Event Iterator handle pointing to it. The architecture follows a traditional client/server paradigm, the Event Distributor being in the role of the central server, which accumulates all event data from all the Event Samplers, shipping them to all Monitoring Tasks requesting events from Event Samplers. An UML diagram showing the event distribution sequence of the previous implementation can be found in figure 3.1.

3.2 API usability

In the previous implementation the API is easy to understand, very generic and unambiguous. On the Event Sampler side there exists a single abstract class EventSampler with two virtual methods startSampling and stopSampling. When the Event Distributor receives the first SELECT request for a certain combination of Event Sampler and selection criteria, it invokes startSampling on the appropriate EventSampler, creates an EventIterator object and returns a reference to the Monitoring Task. As soon as this Event Iterator is destroyed, stopSampling will be invoked on the EventSampler. In the users’ startSampling implementation they need to create a new thread, which pushes
events to the Event Distributor's event buffer, in the `stopSampling` implementation, users need to take care of stopping this thread. In the Monitoring Tasks, users retrieve an Event Iterator object by invoking the `SELECT` method. They can then use this Event Iterator in order to retrieve events either in synchronous or asynchronous mode using `nextEvent` or `tryNextEvent`.

### 3.3 Drawbacks

One can see that there is a scalability problem in the previous distribution approach. As the number of Event Samplers and Monitoring Tasks per IPC partition grows, the bandwidth and computing power of the machine, running the Event Distributor will be exhausted, because all Event Samplers and Monitoring Tasks use a single machine for event distribution, thus sharing the bandwidth and performance of this single machine. Apart from that, thread management in the Event Sampler application remains the user's responsibility due to the push model being used, which frequently lead to problems according to past's experiences. Regarding error recovery and fault tolerance there is another problem, as a silently attached Monitoring Task will not notice the crash of an Event Sampler application. Not knowing if this happened due to a temporary network outage or an Event Sampler crash, it will just stop receiving event data.
Chapter 4

Reimplementation

4.1 Architectural Design

As already mentioned in chapter 3.1, the previous implementation follows the classical Client/Server paradigm, thus all Monitoring Tasks sharing the bandwidth of the Event Distributor and all Monitoring Tasks producing load in the Event Distributor machine. Having this in mind, one can come up with another solution, that follows the Peer-To-Peer paradigm, i.e. rather than using the Event Distributor as a central server for data distribution, Monitoring Tasks and Event Samplers can be directly connected to each other. In figure 4.1 one can see how this can be done considering the case of two Monitoring Tasks and two Event Samplers. When having more Monitoring Tasks connected to one Event Sampler, one cannot easily apply this idea anymore, as this would lead to a bottleneck in the Event Sampler application, as one can see in figure 4.2. The central notion that has been put forth in the course of this thesis in order to avoid this problem is the idea of a Monitoring Tree, which only connects the root Monitoring Task of a tree to the Event Sampler application, performing the real distribution of event data in the branches of the tree. Each Monitoring Task sends the event data received to a certain number of children Monitoring Tasks that are attached to it. The Event Distributor - which will now be named Conductor - as it does not really distribute events anymore, takes on the administration of this tree. A simple example of a Monitoring Tree can be seen in figure 4.3.

Summarizing this idea one can boil it down to the following statement: While the current implementation shares the bandwidth and computing power of the machine running the Event Distributor application among all Monitoring Tasks

![Figure 4.1: Getting rid of the bottleneck; n Monitoring Tasks/n Event Samplers](image-url)
and accumulates the load raised by the Monitoring Tasks in the Event Distributor machine, the reimplementation will share the load raised by Monitoring Tasks among them and accumulate the bandwidth and computing power of all machines running Monitoring Tasks. Although this seems to be a nice solution for the bottleneck caveat, it also creates new problems. Depending on the degree of nodes in the Monitoring Tree and the depth from the root Monitoring Task to the leafs, leaf Monitoring Tasks will experience a slight delay in the reception of events, so the latency between the sampling of events and the reception in leaf Monitoring Tasks might be potentially higher than in the previous implementation. In practice however for a high number of Monitoring Tasks connected to a single Event Sampler, the latency for leaf Monitoring Tasks will be even lower than in the previous implementation, as here the Event Distributor has to ship events multiple times (once for every Monitoring Task). Because of this the latency should increase linearly with the number of attached Monitoring Tasks, while the latency induced by the Monitoring Tree for leaf Monitoring Tasks scales in a k-logarithmic manner, depending on the degree k of the Monitoring Tree. Apart from that, latency in general is not considered to be a criteria for the Event Monitoring System according to [9]. Another problem is that while in the previous implementation Monitoring Tasks are pure clients and exiting Monitoring Tasks will not influence the system, in the reimplementation Monitoring Tasks are clients of Event Sampler applications and possibly servers for attached child Monitoring Tasks. Apart from that Monitoring Tasks may run on potentially unstable user machines. An exit of a Monitoring Task therefore has to be handled in the right manner. One can distinguish between three different cases.
Exit/\textbf{Crash of an internal leaf Monitoring Task} \hspace{1em} This is a trivial case, as leaf Monitoring Tasks do not cover a server role for any children Monitoring Tasks as they have none. The parent sending events to the exiting Monitoring Task will notice the exit and just delete its subscription. It will then notify the Conductor about the exit, so the Conductor’s view of the Monitoring Tree remains consistent. A further rearrangement of the Monitoring Tree is not necessary in this case.

Exit/\textbf{Crash of an internal Monitoring Task with child(ren)} \hspace{1em} In this case, the exiting Monitoring Task has children, so it covers a server role for them. The parent sending events to the exiting Monitoring Task will notice the exit and notify the Conductor about it. Having global knowledge about the whole Monitoring Tree, the Conductor can then rearrange it, assigning one of the children of the exited Monitoring Task to its parent Monitoring Task and attaching remaining children to leaf Monitoring Tasks.

Exit/\textbf{Crash of a root Monitoring Task} \hspace{1em} This is the most critical case, as the exiting root Monitoring Task used to be connected directly to the Event Sampler application, which runs on a machine inside the data flow chain of the detector. If the root Monitoring Task of a Monitoring Tree exits, the Event Sampler application will notice this and notify the Conductor about it. The Conductor then simply selects one of the children of the exited Monitoring Task and allocates it as new root of the Monitoring Tree, while attaching the remaining children to leaf Monitoring Tasks. The sampling process can proceed, Monitoring Tasks will not notice the exit, apart from a slight delay, while a new root of the Monitoring Tree is chosen. The Event Sampler needs to swap the root subscription with the new root Monitoring Task.

\section*{4.2 API Usability}

Emphasis should be put on the fact, that there is much effort being invested in the usability and convenience of the Event Monitoring Framework and in genericity considerations. As all of this has already been explored in the previous implementation and as there are existing applications for the previous framework, the reimplementation follows the currently used API whenever possible, while trying to simplify the usage of the framework. One of the most important things that have been learned in the context of this thesis is, that a clean and simple API will be of major importance for the prevention of user errors and API misunderstandings. This has been achieved by hiding any unessential detail of implementation, resulting in an API, that to the largest possible extent cannot be misused, even when done on purpose. It is clear, that this is a great challenge, considering the fact that none of these usability deliberations shall influence the genericity of the framework. An important aspect of any API consideration is the hiding of any memory management from the users. As this, especially in C++ is a common mistake, it is of major importance to hide memory management inside the framework, the user not being aware of it. In order to achieve this, proxy classes have been implemented, which will automatically
clean up memory when they get out of scope. Whenever a user’s buffer is used for an API function, it must be guaranteed, that the buffer may be freed or altered, after the function has returned. Especially in the pushEvent function, this behavior is of major importance.

A distribution sequence diagram as well as a collaboration diagram using the UML notation, showing the API of the Event Monitoring System reimplementation can be seen in figure 4.4 and figure 4.5.

Figure 4.4: Distribution sequence diagram (UML notation)

### 4.2.1 Push vs. Pull Model

While the previous implementation follows the push model in the user’s Event Sampler code (i.e. user threads have to actively push events to event channels), the reimplementation changes this to a pull model, in order to hide thread details from the user. The user may implement a function `sampleEvent`, which will be called from an internally managed thread whenever an event should be sent to a Monitoring Task. While this is certainly more convenient and less error prone for the average user, there have been complaints by some Event Sampler developers, who need to have more control over the event sampling process. This has led to a split API on the Event Sampler side, providing a very simple and clean pull model API for users who do not wish to care about thread management, but also offering a push model API, which can be used with a custom thread for pushing events to the event channels. In this case it is the user’s responsibility to take care of thread management and synchronization.
4.2.2 Genericity considerations and design pattern

As the reimplementation in case of the pull model offers automatic thread management to the users, the Event Sampler side of the framework has become much more complex than it used to be in the previous implementation. While the Event Sampler API of the previous implementation, which is described in chapter 3.2 used to be really simple and very generic it exhibits the drawbacks mentioned earlier in chapter 3.3. In the reimplementation Event Samplers need to do much more than simply passing the invocation of startSampling and stopSampling to the users implementation of these virtual methods. In case of the pull model the Event Sampler has to automatically manage a pool of threads for every selection criteria being sampled, each thread calling a method that needs to be implemented by the user. So in case of the pull model, a custom user object has to be created dynamically by the framework whenever a new event channel with a new pull thread needs to be created. As the exact object type is not known beforehand the factory method pattern has been used for the Event Sampler API design. A factory object PullSamplingFactory creates an object of type PullSampling which is an abstract class, that needs to be implemented by the user of the framework. The users implementation of sampleEvent will then be called by the pull sampling thread. In order to unify the API on the Event Sampler side the factory design pattern was also used for push model, providing an abstract class PushSampling. Whenever an instance of the users implementation of PushSampling is created by PushSamplingFactory, it is as-
4.2 API Usability

assumed that the user will start a new thread pushing events to an event channel object which will then take care of proper distribution of events. One possibility to implement this design pattern would be the usage of template factory classes (Note: the framework only needs to support C++ Event Samplers). The user then might implement its custom PullSampling or PushSampling classes, create an appropriate templated factory object and pass this factory object to the constructor of EventSampler. An example implementation outline of this template-based API can be seen in figure 4.6. While in this approach users only need to provide an implementation of their custom PullSampling class, it has a structural disadvantage: Users are not free to pass custom parameters to the constructor of PullSampling like they could do in the previous implementation, as these now would be fixed by the implementation of the startSampling method in PullSamplingFactory. There is however the need for custom parameters, as the creation of an PullSampling or PushSampling object usually needs to initialize the data flow chain for actual event sampling. In order to provide custom parameters rather than using a template factory class, abstract factory classes have been used in the implementation, which need to be inherited by custom user factory classes. The actual API of the current version of the implementation will be described in more detail in section 4.2.3.

Figure 4.6: Outline of a template class factory API with some pseudocode

\[
\text{abstract class PullSampling} \\
\quad \text{virtual PullSampling(const SamplingAddress &sa,} \\
\quad \quad \text{const SelectionCriteria &sc) \{ ; \}} \\
\quad \text{virtual void sampleEvent(} \\
\quad \quad \text{emon::EventChannel & ec) = 0;} \\
\quad \text{virtual ~PullSampling()} \\
\quad \text{\{ ; \}} \\
\text{\}} \\
\text{template <class T=PullSampling> class PullSamplingFactory} \\
\quad \text{PullSamplingFactory()} \{ ; \} \\
\quad \text{T * startSampling(const SamplingAddress & sa,} \\
\quad \quad \text{const SelectionCriteria & sc) \{} \\
\quad \quad \quad \text{return new T(sa, sc);} \\
\quad \text{\}} \\
\text{// USER Code} \\
\text{class MyPullSampling : PullSampling} \\
\quad \text{PullSampling( const SamplingAddress &sa,} \\
\quad \quad \text{const SelectionCriteria &sc) \{} \\
\quad \quad \quad \text{// do INITIALIZATION} \\
\quad \} \\
\quad \text{void sampleEvent(emon::EventChannel & ec) \{} \\
\quad \quad \text{// PUSH EVENT TO \ ec} \\
\quad \} \\
\quad \text{~PullSampling()} \\
\quad \text{\{ // do CLEANUP} \\
\quad \} \\
\text{main()} \\
\quad \text{\{} \\
\quad \quad \text{EventSampler sampler( //PARTITION*, /\DORENS*/, new PullSamplingFactory<MyPullSampling>();} \\
\quad \} \\
\]
4.2.3 System APIs

The API description of the Event Monitoring reimplementation presented in this chapter 4.2.3 is an updated and extended version of the API chapter published in the High Level Design document [12]. This document has been published in 2005 at CERN in order to receive user’s feedback on the new API.

4.2.3.1 Sampling Address and Selection Criteria

According to [9], the Event Monitoring System shall allow users to specify the source of events that shall be sampled and certain characteristics all sampled events need to fulfill. Two abstractions used in order to fulfill these requirements are the sampling address and the selection criteria. A sampling address is a simple sequence of address keys and values, which may be passed to Event Sampler applications in order to register its address and to Monitoring Tasks in order to specify the address of the Event Sampler to sample events from.

In order to define which events are interesting for them, users may pass a selection criteria to the Monitoring Tasks on startup. Depending on the position in the data flow chain, users may select events according to different characteristics. At ROD level, the level 1 trigger type and the detector type may be specified. At ROS level, users may perform a selection using the level 1 trigger type and the detector type in the read out buffer (ROB) header. Finally at EB level, users may use selection criteria using a level 1 trigger type and a level 2 trigger info. So at all stages a two-word selection criteria is sufficient.

A one-word selection criteria consists of a so-called masked value. Masked values contain a simple long value and a boolean ignore flag, telling whether this value should be ignored (masked). When the ignore flag is set to true, every event will match the selection criteria, independent from the long value specified. Apart from masked values, the selection criteria contains another long parameter statistics, which may be used to specify how many events should be sampled. A value of $x$ shall result in every $x^{th}$ event (in other words $\frac{100}{x}$% of all events) being sampled. The precise definitions of the SamplingAddress and SelectionCriteria structs can be seen in figure 4.19.

4.2.3.2 Event Sampler API

In order to provide a simple API for the common user while at the same time delivering a powerful API to users who want to have full control about the sampling process, the API has been split into two parts as described in section 4.2.1. Both parts of the API will be discussed in detail in the following paragraphs.

**Pull model API** The user will have to provide the main entry point of the Event Sampler application by instantiating a class which is called **EventSampler**. The **EventSampler** constructor expects a minimum of three arguments as it is shown in figure 4.7. The first parameter defines the TDAQ partition, to which the Event Sampler will belong. The second one provides the sampling address, for which this Event Sampler will be responsible. All the
sampling requests with the same sampling address will be forwarded by the Conductor sub-system to this Event Sampler. The third argument is a pointer to the user’s sampling factory object, that will be used for event channel creation. This parameter is used to hide thread management aspects from application developers in order to simplify their work and to have Event Sampler implementations less error prone, as mentioned in chapter 4.2.1. The optional fourth argument specifies the maximum number of event channels this Event Sampler might create. One so-called event channel will be created for every selection criteria being sampled by this Event Sampler application. As one Monitoring Task connection will be allowed per event channel, this maximum number will influence the distribution performance of the Event Sampler application. If not specified, the default value of 100 will be used. As the EventSampler class is common among the push and pull API, in this section the constructor suitable for the pull model will be used.

Having made the considerations in section 4.2.2 the factory design pattern,

```cpp
class EventSampler
{
public:
    EventSampler ( /* FULL CONSTRUCTOR DETAILS*/ );
    EventSampler( const iP*partition & partition,
                  const SamplingAddress & address,
                  PullSamplingFactory * factory,
                  unsigned long max_channels = 100 );
    "EventSampler();
};
```

which is used by the EventSampler class requires an application developer to implement two abstract interfaces. In the case of the Event Sampler these interfaces are called PullSampling and PullSamplingFactory, which are shown in figure 4.8. In order to implement the PullSamplingFactory interface an Event

```cpp
struct PullSampling
{
public:
    virtual "PullSampling();
    virtual void sampleEvent( EventChannel & cc )=0;
};

struct PullSamplingFactory
{
public:
    virtual PullSampling * startSampling(
        const SelectionAddress & address,
        const SamplingCriteria & criteria)
        throw BadAddress, BadCriteria = 0;
    virtual "PullSamplingFactory();
};
```

Sampler developer has to inherit his own class from the PullSamplingFactory and implement the startSampling virtual function. A possible implementation is shown in figure 4.9.
An implementation of the `startSampling` method has to return an instance
of the user specific class, which must implement the `PullSampling` interface. An
instance of the `MyPullSampling` class is responsible for performing the actual
event sampling, i.e. for the interaction with the data flow system. The
`PullSamplingFactory` is necessary to give some flexibility in defining signatures
for the `PullSampling` object constructor to the developer as mentioned in
section 4.2.2. Figure 4.10 shows a possible implementation of the `PullSampling`
interface, which requires the `SelectionCriteria` parameter and may take any
additional custom parameters. An Event Sampler developer is free to define
any parameters, which he may need to be provided for the `PullSampling`
object. Please note, that there is a bijective relationship between instances of
`PullSampling` and threads sampling events for a selection criteria respectively
event channels.

There are several things, which have to be done for a proper implementa-

Figure 4.10: Example of the `PullSampling` interface implementation (with some
pseudo code)
```cpp
class MyPullSampling : public PullSampling
{
    public:
        MyPullSampling( const SelectionCriteria & criteria, ANY_CUSTOM_ARGUMENTS )
        {
            unsigned long * buffer = new unsigned long(EVENT_SIZE);
            INITIALIZE_DATAFLOW_SYSTEM WITH criteria AND CUSTOM_ARGUMENTS
        }
        ~MyPullSampling()
        {
            delete[] buffer;
        }

        void sampleEvent( EventChannel & ec )
        {
            long event_size = READ_EVENT_TO_BUFFER( buffer );
            ec.pushEvent( buffer, event_size );
        }
};
```
tion of the abstract PullSampling class. It is assumed that all the necessary initialization, which has to be done to prepare for the event sampling process will be performed in the constructor of the user defined MyPullSampling class. When the destructor of that class is called, this indicates that event sampling is not necessary anymore and MyPullSampling has to perform a proper clean-up procedure. The main working method of the MyPullSampling class is the sampleEvent function, which is responsible for reading an appropriate event from the data flow system and pushing it to the EventChannel by using the pushEvent function. This technique is used to avoid complicated memory management in the user code, which would appear if the sampleEvent function was declared as returning the event. By using an EventChannel object for pushing the event to Monitoring Tasks, the framework can guarantee, that the buffer used for passing the event to the event channel may be freed or reused after pushEvent returns. This wouldn’t be possible if users would return an event from sampleEvent. Finally figure 4.11 shows an example of the main function for the Event Sampler application. The function wait() of class EventSampler might be used to block the current thread until the event sampling process is stopped with the function stop(). If initialization of the Event Sampler fails for some reason, CannotInitialize is thrown from the constructor of class EventSampler.

Figure 4.11: Example of the main function for an Event Sampler application (with some pseudocode)

```c
void stop_sampling
{
    sampler->stop();
}

int main()
{
    IPCPartition partition = PARTITION;
    max_channels = 100;
    SamplingAddress address = ADDRESS;
    emon::EventSampler temp(partition, address, new MyPullSamplingFactory(), max_channels);
    sampler = &temp;
    sampler->wait;
    return 0;
}
```
Push Model API While the pull model will be suitable for many users, some users might need more control over the event sampling process. Just like in the case of the pull model, the user will have to provide the main entry point of the Event Sampler application by instantiating a class called EventSampler. The EventSampler constructor expects a minimum of three arguments as it is shown in figure 4.12. The first parameter defines the TDAQ partition, to which the Event Sampler will belong to. The second one provides the sampling address, for which this Event Sampler will be responsible. All the sampling requests with the same sampling address will be forwarded by the Conductor sub-system to this Event Sampler. The third argument is a pointer to the user’s sampling factory object, that will be used for event channel creation just like its done in the pull model API. The optional fourth argument specifies the maximum number of event channels this Event Sampler might create. One so-called event channel will be created for every selection criteria being sampled by this Event Sampler application. As one Monitoring Task connection will be allowed per event channel, this maximum number will influence the distribution performance of the Event Sampler application just as it does in the pull model API. Again, if not specified the default value of 100 will be used. As the EventSampler class is common among the Push and Pull API, in this section the constructor suitable for the push model will be used. Just like in the case of the pull model,

Figure 4.12: Event Sampler class (Push API)

class EventSampler
{
public:
    EventSampler ( /* PULL CONSTRUCTOR DETAILS*/ );
    EventSampler ( const TDAQ::Partition & partition, 
                  const SamplingAddress & address, 
                  PushSamplingFactory * factory, 
                  unsigned long max_channels = 100 )
        : EventSampler( )
    }
};

the push model API uses the factory design pattern, which requires implementation of two interfaces. In the case of push model these interfaces are called PushSampling and PushSamplingFactory, which are shown in figure 4.13. In order to implement the PushSamplingFactory interface an Event Sampler developer has to inherit his own class from PushSamplingFactory and implement the startSampling virtual function. A possible implementation is shown in figure 4.14.

An implementation of the startSampling method has to return an instance of a user specific class, which must implement the PushSampling abstract interface. An instance of the PushSampling class is responsible for performing the actual event sampling, i.e. for the interaction with the data flow system. The PushSamplingFactory is necessary to give some flexibility in defining the signature for the PushSampling object constructor to the developer. Figure 4.15 shows a possible implementation of the PushSampling interface, which requires the SelectionCriteria parameter and - again - may take any additional custom arguments. Again at this point an Event Sampler developer is free to define any custom parameters, which he may need to be provided for
4.2 API Usability

Figure 4.13: PushSampling and PushSamplingFactory interfaces

```cpp
struct PushSampling
{
    public:
        virtual ~PushSampling();
};

struct PushSamplingFactory
{
    public:
        virtual PushSampling * startSampling(
            const SelectionAddress & address,
            const SamplingCriteria & criteria,
            EventChannel * ch)
            throw BadAddress, BadCriteria = 0;
        virtual ~PushSamplingFactory();
};
```

Figure 4.14: A possible PushSamplingFactory implementation (with some pseudo code)

```cpp
class MyPushSamplerFactory : public emon::PushSamplingFactory
{
    public:
        emon::PushSampling * startSampling( const emon::SamplingAddress & ,
                                            const emon::SelectionCriteria & criteria,
                                            emon::EventChannel * channel )
        {
            return new MyPushSampler( criteria, channel, ANY_CUSTOM_ARGUMENTS);
        }
};
```

a new PushSampling object. There are several things, which have to be done for a proper implementation of PushSampling class. All necessary initialization which needs be done in order to prepare for the event sampling process, as well as the initiation of the sampling thread, has to be performed in the constructor of the user defined MyPushSampling class. When the destructor of that class is called, this indicates that the sampling thread is not needed anymore and the sampling thread has to perform a proper clean up procedure. All thread issues and actual event distribution is left to the user. Event distribution can be done by calling the function pushEvent of the EventChannel object. Finally figure 4.16 shows an example of the main function for the Event Sampler application. The function wait() of class EventSampler might be used to block the current thread until the event sampling process is stopped with function stop(). If initialization of the Event Sampler fails for some reason, CannotInitialize is thrown from the constructor of class EventSampler.
Figure 4.15: Example of the PushSampling interface implementation (with some pseudo code)

class MyPushSampler : public emon::PushSampling, 
  public QThread 
{
  public:
    emon::EventChannel * channel_;

  MyPushSampler( const emon::SelectionCriteria &sc, 
    emon::EventChannel * channel, ASY_CUSTOM_ARGUMENTS )
  : channel_( channel )
  {
    INITIIZE DATA FLOW WITH CUSTOM ARGUMENTS
    // this will call run_undetached in a separate thread
    start_undetached();
  }

  void * run_undetached( void * )
  {
    while( ! m TERMINATED )
    {
      RETRIEVE EVENT
      channel_->pushEvent(EVENT, SIZE);
      return 0;
    }
  }

  ~MyPushSampler()
  {
    TERMINATE AND CLEAN UP
  }
};

Figure 4.16: Example of the main function for Event Sampler application (with some pseudocode)

STOP_SAMPLING
{
  sampler->stop();
}

int main()
{
  PCP::Partition partition = PARTITION;
  max_channels = 100;
  SamplingAddress address = ADDRESS;
  emon::EventSampler temp(partition, address, new MyPushSamplingFactory(), max_channels);
  sampler = new sampler;  // terminate
  sampler->wait;
  return 0;
}

4.2.3.3 Monitoring Task API

The main entry point for a Monitoring Task application is the select function, which is defined in the emon name space as shown in figure 4.17. The argument buffer_limit specifies the maximum amount of events the buffer in this Monitoring Task can hold. If no buffer_limit is specified, a default maximum buffer size of 1000 events is used. The parameter buffer_limit will affect memory usage of the Monitoring Task, as it is the maximum number of events
a Monitoring Task will cache for the local user (until events are processed) and possibly child Monitoring Tasks (until events are forwarded).

This function allocates an instance of the **EventIterator** class, which a de-

```cpp
class EventIterator
{
  public:
    EventIterator() : it_(nullptr),
    p_(nullptr),
    address_(nullptr),
    selectionCriteria_(nullptr),
    buffer_limit_(1000)
    {
      // Code...
    }

  ~EventIterator()
  {
    // Code...
  }

  // More code...
}
```

Figure 4.17: Monitoring Task entry point

A developer can use to retrieve events, that have been sampled from the sampling address and satisfy the selection criteria. This function may throw several exceptions in case of either address or criteria is invalid, the Conductor is not available, the Event Sampler does not accept any more Monitoring Tasks (maybe because the maximum number of allowed event channels would be exceeded) or this Monitoring Tasks cannot be added to the Monitoring Tree for any reason.

As shown in figure 4.18, the **EventIterator** class provides two functions to retrieve events. The **nextEvent** function returns a new event or suspends for timeout milliseconds before throwing the **NoMoreEvents** exception in case there are no more events in the event buffer. If no timeout is specified, it will block the calling thread until a new event is available, waiting infinitely. The **tryNextEvent** function returns a new event if available or immediately throws a **NoMoreEvents** exception in case the underlying event buffer is empty and may be used for asynchronous event retrieval. The **availableEvents** function returns the number of events currently available in the Monitoring Task's buffer showing the current buffer fill rate in the Monitoring Task. The **eventsDropped** function indicates whether this Monitoring Task has dropped events due to an overfull buffer. This parameter might be interesting for users that rely on the fact that really all events arrive at the Monitoring Task. As the underlying CORBA oneway function calls used for event distribution only offer At-Most-Once semantics, users can not be sure that they receive all events even if the **eventsDropped** function returns zero. Events can be lost either due to network congestion or in earlier layers of the data flow chain, which can not be influenced by the emon package. Please note that in agreement with [9] event delivery is not guaranteed for any particular event.
Figure 4.18: Event Iterator interface

class EventIterator
{
   public:
      EventIterator();
      Event * nextEvent ( unsigned long timeout = 0 ) throw SoMoreEvents;
      Event * tryNextEvent() throw SoMoreEvents;
      unsigned int availableEvents();
      bool eventsDropped();
};
4.3 Implementation details

4.3.1 IDL declaration

Figure 4.19 shows the IDL declaration of the Event Monitoring System.

Figure 4.19: The Event Monitoring System interfaces (OMG IDL)

```idl
#include <guide/idl>

module EventMonitoring {
  typedef sequence<short> long EventFragment;
  typedef sequence<EventFragment> Event;

  struct MaskedValue ( long value; boolean ignore; )
  struct SelectionCriteria {
    MaskedValue detector_type;
    MaskedValue trigger_type;
    MaskedValue trigger_info;
    MaskedValue state_word;
    long statistics;
  }
  typedef sequence<SelectionCriteria> SamplingAddress;
  struct AddressComponent ( string key; string value; )
  enum Direction { Forward, Backward }

  interface EventMonitor {
    event void push_event ( in Event e );
    void get_children ( out MonitorList children );
    void add_child ( in EventMonitor monitor );
    void remove_child ( in EventMonitor monitor );
    void sample_exit ( );
  }

  interface EventSampler : instrumentation {
    void ping ( );
    void connect_monitor ( in SamplingAddress address,
                          in SelectionCriteria criteria,
                          in EventMonitor monitor )
      raises ( ResourceError, BadAddress, BadCriteria, AlreadyConnected );
    void replace_monitor ( in SamplingAddress address,
                          in SelectionCriteria criteria,
                          in EventMonitor monitor )
      raises ( ResourceError, BadCriteria, NotConnected );
    void disconnect_monitor ( in SamplingAddress address,
                            in SelectionCriteria criteria,
                            in EventMonitor monitor )
      raises ( ResourceError, BadCriteria, NotConnected );
    void adapt_sampling_rate ( in SamplingAddress address,
                               in SelectionCriteria criteria,
                               in EventMonitor monitor )
      raises ( ResourceError, BadCriteria, NotConnected );
    void get_monitors ( out EventMonitor monitors );
  }

  interface EventFilter {
    const string name = "Filter";
    void connect_filter ( in SamplingAddress address,
                         in EventFilter filter )
      raises ( AlreadyExists );
    void disconnect_filter ( in SamplingAddress address,
                            in EventFilter filter )
      raises ( NotConnected );
    void add_filter ( in SamplingAddress address,
                     in SelectionCriteria criteria,
                     in EventFilter filter )
      raises ( NotConnected );
    void remove_filter ( in SamplingAddress address,
                        in SelectionCriteria criteria,
                        in EventFilter filter )
      raises ( NotConnected );
    void adapt_sampling_rate ( in SamplingAddress address,
                               in SelectionCriteria criteria,
                               in EventFilter filter )
      raises ( NotConnected );
  }
}
```
4.3.2 Thread Safety Considerations

4.3.2.1 Event Forwarding

A separate single thread for the sake of event forwarding is needed in Monitoring Tasks, because otherwise the rate at which a Monitoring Task can receive events would synchronize with the speed at which child Monitoring Tasks can receive events, leading to a loss of performance with every additional level of child Monitoring Tasks present. In case of the exit of a child Monitoring Task, the Conductor will be notified about this, calling back the parent Monitoring Task, telling it whether to remove the Monitoring Task or add another one. As this call-back from the Conductor has to be processed by a Monitoring Task while it is still waiting for the call to the Conductor to finish in case a child Monitoring Task has failed, it is clear that IPCs (respectively CORBA POAs) multi-threaded behavior is needed here, enabling the call-back to the Monitoring Task to be processed without resulting in a deadlock that would appear with single-threaded behavior. When using single-threaded behavior in Monitoring Tasks one can look at this problem as sort of a distributed friendly semaphore semantics that would be required in order to allow one function of a Monitoring Task to be called back (which can be seen as sort of a thread reentrance) while the function causing the call-back still waits to complete and holds the object's lock. Because of the synchronous remote service invocation paradigm used by CORBA's two-way functions, one might look at a remote method invocation as a thread migrating to another machine (the thread in the caller blocks until the thread in the callee returns, so there is an implicit mutual exclusion between thread activities).

In order to decouple the event forwarding from the event receiving process, another buffer for events waiting to be forwarded to child Monitoring Tasks is needed. There is no alternative to this second buffer, as without it, one would have to forward an event directly after reception, leading again to a speed coupling of the rate at which Monitoring Tasks can receive events and the rate at which its child Monitoring Tasks may receive events. Another solution might be the creation of a new thread for every incoming event, the new thread forwarding a single event to the children and then being destroyed again. In case of slow Monitoring Tasks, the number of threads waiting for their event being forwarded might become huge, which might cause problems. In the current version of the reimplementation decision has been made to implement a single separate event forwarding thread which will be created with the first child Monitoring Task subscription and automatically destroyed when the last subscription of a child Monitoring Task is removed. Two buffers are used for holding smart pointers to events that still need to be forwarded and those that still need to be processed by the user. This smart pointer solution will be discussed in more detail in chapter 4.3.3.

4.3.2.2 Thread synchronization

Especially in the case of Monitoring Tasks, where several threads read and write to and from two ends of a single buffer as described in 4.3.2.1, thread synchronization is of major importance. The C++ part of the reimplementation
uses condition variables and simple mutex objects for synchronization, which are provided by the OWL-component of the ATLAS Online Software. The Java part of the reimplementation (Java Monitoring Task implementation and event iterator class) uses a custom semaphor class, that has been implemented on top of the standard monitor implementation JAVA provides. As all CORBA servant implementations except for the Conductor implementation use multi-threaded behavior thread synchronization had to be used whenever shared data are accessed.

### 4.3.3 Smart Pointer Memory Management

In an early version of the reimplementation of the Event Monitoring System done in the context of this thesis, Monitoring Tasks put any events they received to their local event buffer where they waited for being processed by the user. A copy of the event has been made in order to put it to a separate forwarding buffer, where events waited to be forwarded to child Monitoring Tasks. While this is a simple solution it naturally proved to be a quite inefficient way to solve the problem of the separate forwarding thread due to the computational overhead induced by the copying of event data (these event data may in case of the Event Builker profile be as large as 2 MB), so the latest version of the reimplementation uses the smart pointer implementation of BOOST in order to avoid event copying. While still having two separate buffers for the event forwarding thread and events that wait for being processed by the user via the `EventIterator`, these buffers now only hold a smart pointer to the actual event in memory, thus leading to an automatic deallocation of event memory as soon as the last reference to an event has been deleted. This solution not only eliminates the copy operation needed in the previous version, but also simplifies memory management for users, as they now receive a smart pointer to an event, relieving them from manually deleting the event when they do not need it anymore.

### 4.4 Scalability

Considering scalability, the Peer-To-Peer paradigm used for the reimplementation offers perfect behavior. Instead of exhausting the resources of the Event Distributor like in the previous implementation, as mentioned in chapter 3.3, adding Monitoring Tasks and Event Sampler applications now results in an increase of computing power and bandwidth of the overall Event Monitoring System, as these are actively used for event distribution. The Event Distributor application, which is now called Conductor to reflect its new role, is not used for communication anymore so the architecture described in chapter 4.1 offers arbitrary scalability (within the limits entailed by CORBA). This will be proven theoretically in chapter 5.3, showing that with only constant increase in the communication complexity of the Monitoring Task applications the communication complexity of the Conductor can be reduced to zero for the communication phase. An important aspect of any architectural design is the load induced in the Event Sampler applications, as they run on machines inside the data flow chain. So it is absolutely crucial that no additional load is put to Event Sampler
applications. Again chapter 5.3 will show, that there is no additional overhead here. The Conductor is not only used for connection and subscription management but also for encapsulation of Event Sampler applications. Monitoring Tasks may only connect to Event Sampler applications by using the Conductor, so the Conductor offers an important line of defense between Event Samplers and Monitoring Tasks.

4.5 Fault Tolerance

As seen in chapter 3.3, fault tolerance was one of the drawbacks of the previous implementation, as a Monitoring Task will not even notice the exit or crash of an Event Sampler application from which it samples events. Offering fault tolerant behavior was one of the major priorities respected in the reimplementation. Every component of the Event Monitoring System can now fail without having influence on the remaining applications, except for those directly attached to the failing one.

4.5.1 Crash of an Event Sampler application

One can easily overcome the drawback of infinitely waiting Monitoring Tasks in case of an Event Sampler crash, even without making use of the architectural design of the reimplementation. Assuming that the Conductor sends ping messages to all Event Samplers at a reasonable frequency, it will notice any crashed Event Sampler. Having noticed and having the knowledge of the whole Event Monitoring system, the Conductor can now easily notify the attached Monitoring Task about this crash, which will spread the news among its children. With this method, the whole Monitoring Tree will notice the exit of an Event Sampler application and can issue an appropriate message to the users Monitoring Task performing a clean and verbose exit.

4.5.2 Crash of a Monitoring Task application

Using the Monitoring Tree maintenance discussion, made earlier in chapter 4.1 and considering the fact, that any Monitoring Task crash will be detected just like a regular exit, the conclusion can be made, that the crash of arbitrary Monitoring Task applications will not influence the stability or performance of the Event Monitoring System. But a minor problem remains. If a root Monitoring Task crashes and the associated event channel never gets to sample an event that matches the appropriate selection criteria and which can be pushed to the Monitoring Task, it will never realize the Monitoring Task application actually crashed, because there is no failing `pushEvent` call. In order to solve this problem which actually has been reported by users, two solutions have been investigated. The first solution is making the event channel save a timestamp with each call to `pushEvent`, and checking the timestamp of the last `pushEvent` call every few seconds. If it comes to the conclusion, that the timespan since the last `pushEvent` exceeds a certain threshold, it sends a ping request to the root
Monitoring Task to see whether it is still alive and running. If it is not, the event channel can force disconnection using the Conductor application. The second solution is the Conductor sending periodical ping requests to all Monitoring Tasks, which generally is a bad idea regarding scalability and message complexity, as the Conductor application can not determine the frequency of normal pushEvent calls (which make ping requests obsolete). In the latest version of the reimplementation the first solution has been implemented and successfully tested.

4.5.3 Crash of the Conductor

The basic problem is that the Conductor application needs to be stateful in order to perform maintenance of the Monitoring Tree thus exhibiting the amnesia problem when the Conductor crashes. In this section several methods to reobtain the conductor state (in terms of information about running Monitoring Task and Event Sampler applications) when a Conductor application restarts after a crash will be discussed. Although the Conductor can determine whether a restart happened after a crash or a normal restart (by checking a simple flag in IS that will be set only on clean exit and reset on start), in the current version of the reimplementation, the Conductor application does not differentiate between these two cases. Even when started after a normal exit, the Conductor will try to obtain the state which is appropriate regarding the Monitoring Tasks and Event Sampler applications currently running in the IPC partition where it was started. There are different approaches for recovering the state after the exit or crash of the Conductor.

4.5.3.1 Event Sampler Beaconing

Event Samplers keep sending their object references to the Conductor in specific intervals, so the Conductor will be able to discover them after having restarted. This recovery method suffers from the high number of messages that need to be exchanged, even if the Conductor is healthy and running.

4.5.3.2 Monitoring Task Beaconing

Monitors keep sending their object references to the Event Sampler in specific intervals, so the Conductor will be able to discover them after having restarted. The drawbacks of this recovery strategy are basically the same as the drawbacks of the afore mentioned method. The number of messages would be even potentially higher, as (at least in running state) there will be at least as many Monitoring Tasks in a partition than Event Sampler applications.

4.5.3.3 Primary/Backup Approach

A heartbeat protocol running inside a farm of Conductors all having the same state will discover crashes. A simple election algorithm like e.g. a distributed
maximum search will pick a new Conductor, that can be installed transparently for the Event Samplers and Monitoring Tasks. State is shared among all conductors in the conductor farm, using the Hot-Standby or Warm-Standby paradigm. The implementation of this solution wouldn't be trivial and require a lot of additional complexity in the Conductor.

4.5.3.4 Backup file

A backup file can be written by the Conductor, being read when it restarts. This solution while at first glance being rather simple suffers from the fact that backing up the state to a file itself is a transaction. When the Conductor crashes while writing its state, some form of recovery has to be done, again exhibiting an amnesia problem well-known from the scope of transactions. Apart from that it is only possible to recover the state of the previously running Conductor, not knowing if this state is valid anymore, because Event Samplers or Monitoring Tasks might have exited in the meantime. Backing up and restoring CORBA object references would be easily possible using IOR reference strings.

4.5.3.5 Published Event Samplers

When all Event Sampler applications are published in the IPC partition, the Conductor can easily check on startup, if there are any running Event Sampler applications. If it discovers any it can obtain the reference to the attached root Monitoring Tasks, using it for reestablishment of a local representation of all nodes in the Monitoring Tree by diving into it in DFS manner (Monitoring Trees are spanning trees!). As the Conductor does not participate in communication between Event Samplers and Monitoring Tasks, its exit or crash does not even influence the distribution of event data. This is the currently used method for recovery after a Conductor crash, because it is superior to all other methods previously discussed. This solution guarantees that on startup the Conductor can restore not only the state of the previous Conductor, but the state that actually fits to the currently running Event Sampler and Monitoring Task applications, even if there have been further exits or crashes in the meantime.

4.6 Congestion Control Scheme

Considering the Event Sampler applications and the attached Monitoring Tasks being totally separated applications running on totally different hardware, there might be the case where event processing in a Monitoring Task takes much more time than event sampling and sending in the Event Sampler application. In this case there will be a buffer overflow in the Monitoring Task, leading to events simply being dropped if the buffer size is exceeded. This of course is not an efficient behaviour and should be avoided. In order to do this, a Monitoring Task application will take notice of a buffer overflow as well as of a buffer underrun and react appropriately to it. On a buffer overflow, it will ask the Event Sampler application to add a slight delay to the event sampling process, until
the buffer empties again. Of course the Monitoring Task will not ask for a slow-
down, if a child Monitoring Task is attached to it, that might process events
at a higher speed. On a buffer underrun, the Monitoring Task, will ask for a
speedup, leading to a decrease of delay in the appropriate event channel. In
a first version this scheme was implemented. Although it actually worked, it
still suffered from a pendulum buffer fill rate in the sense, that after a buffer
overflow it emptied totally, filled again totally and so on, which lead to a high
number of sampling rate adaptation requests.
In order to overcome this deficiency, a congestion control scheme similar to that
used by TCP has been implemented in the latest version of the reimplemen-
tation. Starting with a minimum initial delay of one nanosecond, event channels
will multiply their delay with a constant with every speedup request, leading
to an exponential increase in the delay value. With every call to slowdown, the
delay is divided by that same constant, leading to an exponential decrease in
the delay value. Monitoring Tasks will check their buffer fill rate with every nth
call to pushEvent (with n being its maximum buffer size divided by 10) and
store this value in a variable last_buffer_rate. If the deviation of the current
buffer fill rate and the last buffer fill rate exceeds a certain constant maximum,
speedup or slowdown will be called. The idea behind the definition of n as the
maximum buffer size divided by 10 is that one can allow more variation in the
buffer fill rate the more buffer space is available.
In the current version of the reimplementation, sampling rate adaptation is only
allowed for root Monitoring Tasks, as they are directly connected to the sam-
pling applications. It might also be useful to allow sampling rate adaptation
for non-root Monitoring Tasks in future version, as there might be the case
when child Monitoring Tasks can receive events faster than one of their parent
Monitoring Tasks actually can do. When allowing speedup requests from child
Monitoring Tasks, one has to make sure that all parent nodes to this Monitoring
Task will not be allowed to issue slowdown requests anymore, as they cannot
decide this on their own, without knowledge of the whole Monitoring Tree. As
it is not clear by now if this feature will be of any use in the final experiment,
the extra complexity an implementation of this would require has not yet been
taken, but it might be worth consideration in one of the future versions.

4.7 Segmented Memory Support

Some users like those in SFI instead of having events in physical contiguous
memory, hold event fragments in different not necessarily contiguous parts of
the memory. In a first version of the reimplementation and in all previous imple-
mentations, when events were pushed to the event channel, all event fragments
needed to be copied to a contiguous memory section, which produced a compu-
tational overhead due to the extra copy operation involved (event size may be
as big as 2 MB in Event Builder profile). In order to solve this problem, two
different pushEvent methods have been implemented in the latest version of the
reimplementation. One for contiguous memory blocks, accepting a simple con-
tiguous long array and another one for memory fragments, accepting an array
of lvalue structures, that may be scattered across memory. Using the second
method signature will eliminate one copy operation in Event Sampler applica-
tions, as events do not have to be copied to contiguous memory chunks when event data are pushed to the framework. On the Monitoring Task side, copy operations can be limited by doing a single copy only in root Monitoring Task, here copying event data to a single contiguous fragment that will be forwarded to child Monitoring Tasks, so they do not have to perform a copy again. In order to support segmented memory events, the event representation in fDL had to be changed from a simple sequence of long values which was used in the previous implementation to a sequence of event fragments, each event fragment itself being a sequence of long values.
Chapter 5

Communication Complexity

5.1 Definitions

In order to quantify the Distributor bottleneck, a theoretical analysis of the communication complexity will be done in this chapter. For this the definitions shown in table 5.1 will be used.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( s )</td>
<td>number of Event Samplers in partition</td>
</tr>
<tr>
<td>( m )</td>
<td>number of Monitoring Tasks in partition</td>
</tr>
<tr>
<td>( \alpha_i )</td>
<td>number of children of Monitoring Task ( i ) ( (i \in {1,\ldots,m}) )</td>
</tr>
<tr>
<td>( D )</td>
<td>maximum number of children (maximum node degree in Monitoring Tree)</td>
</tr>
<tr>
<td>( \beta_i )</td>
<td>number of criteria (event channels) in Event Sampler ( i ) ( (i \in {1,\ldots,s}) )</td>
</tr>
<tr>
<td>( n_{ij} )</td>
<td>number of sampled events for event channel ( j ) in Event Sampler ( i ) ( (j \in {1,\ldots,\beta_i}) )</td>
</tr>
<tr>
<td>( s_{ijk} )</td>
<td>size of event ( k ) in event channel ( j ) in Event Sampler ( i ) in bytes ( (k \in {1,\ldots,n_{ij}}) )</td>
</tr>
<tr>
<td>( N_i )</td>
<td>number of sampled events in Monitoring Task ( i )</td>
</tr>
<tr>
<td>( S_{ik} )</td>
<td>size of event ( k ) in Monitoring Task ( i ) in bytes ( (k \in {1,\ldots,N_i}) )</td>
</tr>
</tbody>
</table>

5.2 Previous Implementation

5.2.1 Event Sampler

**Lemma 1.** Using the definitions of table 5.1, the communication complexity (considering sent and received bytes) of the Event Sampler \( i \) for \( i \in \{1,\ldots,s\} \) is in

\[
O(\max_{j \in \{1,\ldots,\beta_i\}} n_{ij} \cdot \beta_i)
\]

**Proof.** Let \( c(\text{start}) \) be the communication complexity of a \text{startSampling} request, \( c(\text{stop}) \) be the complexity of a \text{stopSampling} request and \( j \in \{1,\ldots,\beta_i\} \). In the course of event sampling, apart from the actual sending of the event bytes

...
the Event Sampler will receive `startSampling` and `stopSampling` requests for every criteria he samples. For the overall communication complexity one gets:

\[ \beta_i \cdot (c(\text{start}) + c(\text{stop})) + \sum_{j=1}^{\beta_i} \sum_{k=1}^{n_{ij}} s_{ijk} \]

c(\text{start}) and c(\text{stop}) are constant

\[ \Rightarrow \exists C_{\text{start/stop}} \in \mathbb{N} : C_{\text{start/stop}} \geq \left\{ \begin{array}{ll} c(\text{start}) \\ c(\text{stop}) \end{array} \right\} \]

All events are smaller than a certain maximum \( S_{i,\text{max}} \)

\[ \Rightarrow \exists S_{i,\text{max}} \text{ with } s_{ij} S_{i,\text{max}} \forall j \in \{1, \ldots, \beta_i\}, k \in \{1, \ldots, n_{ij}\} \]

\[ \Rightarrow \beta_i \cdot (c(\text{start}) + c(\text{stop})) + \sum_{j=1}^{\beta_i} \sum_{k=1}^{n_{ij}} s_{ijk} \]

\[ \leq \beta_i \cdot (2 \cdot C_{\text{start/stop}}) + \sum_{j=1}^{\beta_i} \sum_{k=1}^{n_{ij}} S_{i,\text{max}} \]

\[ = \beta_i \cdot (2 \cdot C_{\text{start/stop}}) + \sum_{j=1}^{\beta_i} n_{ij} \cdot S_{i,\text{max}} \]

\[ \leq \beta_i \cdot (2 \cdot C_{\text{start/stop}} + \beta_i \cdot S_{i,\text{max}} \cdot \max_{j \in \{1, \ldots, \beta_i\}} n_{ij}) \]

\[ = \beta_i \cdot (2 \cdot C_{\text{start/stop}} + S_{i,\text{max}} \cdot \max_{j \in \{1, \ldots, \beta_i\}} n_{ij}) \]

\[ \leq \beta_i \cdot (C_1 + C_2 \cdot \max_{j \in \{1, \ldots, \beta_i\}} n_{ij}) \text{(for some constants } C_1 \text{ and } C_2) \]

\[ \Rightarrow \text{The communication complexity is in} \]

\[ O\left( \max_{j \in \{1, \ldots, \beta_i\}} n_{ij} \cdot \beta_i \right) \]

\[ \square \]

5.2.2 Monitoring Task

**Lemma 2.** The communication complexity of the Monitoring Task \( i \) is in

\[ O(N_i) \]

**Proof.** Each Monitoring Task invokes `select` and `stopSampling` and receives each event once. Using `c(\text{start})` and `c(\text{stop})` just like in Lemma 1, for the communication complexity one gets:

\[ c(\text{start}) + c(\text{stop}) + \sum_{k=1}^{N_i} S_{ik} \]
c(start) and c(stop) are constant

\[\Rightarrow \exists C_{start/stop} \in \mathbb{N} : C_{start/stop} \geq \begin{cases} c(start) \\ c(stop) \end{cases} \]

All events smaller than a certain maximum \( S_{i,\text{max}} \)

\[\Rightarrow \exists S_{i,\text{max}} \text{ with } S_{ik} \leq S_{i,\text{max}} \forall k \in \{1, \ldots, N_i\} \]

\[\Rightarrow c(start) + c(stop) + \sum_{k=1}^{N_i} S_{ik} \leq 2 \cdot C_{start/stop} + \sum_{k=1}^{N_i} S_{i,\text{max}} \]

\[= 2 \cdot C_{start/stop} + N_i \cdot S_{i,\text{max}} \]

\[\Rightarrow \text{The communication complexity is in } O(N_i) \]

\[\square\]

5.2.3 Distributor

**Lemma 3** (Initialization and Shutdown). *The communication complexity of initialization and shutdown of the distributor is in*

\[O(m)\]

**Proof.** For every Monitoring Task the Distributor receives a `select` request during initialization and a `stopSampling` request during shutdown. Using \(c(start)\) and \(c(stop)\) just like in Lemma 1, one gets:

\[m \cdot (c(start) + c(stop))\]

\(c(start)\) and \(c(stop)\) are constant

\[\Rightarrow \exists C_{start/stop} \in \mathbb{N} : C_{start/stop} \geq \begin{cases} c(start) \\ c(stop) \end{cases} \]

\[\Rightarrow m \cdot (c(start) + c(stop)) \leq m \cdot 2 \cdot C_{start/stop} \]

\[\Rightarrow \text{The communication complexity is in } O(m) \]

\[\square\]

**Lemma 4** (Communication). *The communication complexity of the Distributor excluding shutdown and initialization is in*

\[O(m \cdot \max_{i \in \{1, \ldots, m\}} N_i)\]
5.3 Reimplementation

Proof. During communication phase, the Distributor receives all events from all event channels and sends events to all Monitoring Tasks, so the communication complexity is:

\[
\sum_{i=1}^{m} \sum_{k=1}^{N_i} S_{ik} + \sum_{i=1}^{s} \sum_{j=1}^{\beta_i} \sum_{k=1}^{n_{ij}} s_{ijk}
\]

Event sizes \(s_{ijk}\) and \(S_{ik}\) are smaller than a certain maximum \(S_{\text{max}}\) \(\forall i, j, k\)

\[
\Rightarrow \sum_{i=1}^{m} \sum_{k=1}^{N_i} S_{ik} + \sum_{i=1}^{s} \sum_{j=1}^{\beta_i} \sum_{k=1}^{n_{ij}} s_{ijk}
\]

\[
\leq \sum_{i=1}^{m} \sum_{k=1}^{N_i} S_{\text{max}} + \sum_{i=1}^{s} \sum_{j=1}^{\beta_i} \sum_{k=1}^{n_{ij}} S_{\text{max}}
\]

\[
= \sum_{i=1}^{m} N_i \cdot S_{\text{max}} + \sum_{i=1}^{s} \beta_i \cdot n_{ij} S_{\text{max}}
\]

The number of received events in all Monitoring Tasks has to be greater or equal than the number of sent events in all sampling channels (greater only in the case when there is at least one Monitoring Tree consisting of more than a single root Monitoring Task)

\[
\Rightarrow \sum_{i=1}^{m} N_i \geq \sum_{i=1}^{s} \sum_{j=1}^{\beta_i} n_{ij}
\]

\[
\Rightarrow \sum_{i=1}^{m} N_i \cdot S_{\text{max}} + \sum_{i=1}^{s} \sum_{j=1}^{\beta_i} n_{ij} S_{\text{max}}
\]

\[
\leq 2 \cdot \sum_{i=1}^{m} N_i \cdot S_{\text{max}}
\]

\[
\leq 2 \cdot S_{\text{max}} \cdot m \cdot \max_{i \in \{1, \ldots, m\}} N_i
\]

\[
\Rightarrow \text{The communication complexity is in}
\]

\[
O(m \cdot \max_{i \in \{1, \ldots, m\}} N_i)
\]

\[
\square
\]

5.3 Reimplementation

Using the notations defined in table 5.1, the following lemmas can be postulated. In order to simplify analysis we assume the case when sampling rate adaptation is turned off in the Conductor application.
5.3.0.1 Event Sampler

**Lemma 5.** The communication complexity of the Event Sampler \( i \) for \( i \in \{1, \ldots, s\} \) is in
\[
O(\max_{i \in \{1, \ldots, s\}} n_{ij} \cdot \beta_i)
\]

**Proof.** This follows immediately from the fact, that the communication scheme in the Event Samplers has not been changed apart from Event Samplers now sending events to a single root Monitoring Task rather than sending it to a central Distributor. So the proof in lemma 1 remains the same. \( \square \)

5.3.1 Monitoring Task

**Lemma 6.** The communication complexity of the Monitoring Task \( i \) for \( i \in \{1, \ldots, m\} \) is in
\[
O(N_i)
\]

**Proof.** The Monitoring Task \( i \) invokes \texttt{start} and \texttt{stop}, receives each event and forwards it to all of its children. So the communication complexity is
\[
c(\texttt{start}) + c(\texttt{stop}) + \sum_{k=1}^{N_i} S_{ik} + \alpha_i \cdot \sum_{k=1}^{N_i} S_{ik}
\]
\( c(\texttt{start}) \) and \( c(\texttt{stop}) \) are constant
\[
\Rightarrow \exists C_{\texttt{start/stop}} \in \mathbb{N} : C_{\texttt{start/stop}} \geq \left\{\begin{array}{l}c(\texttt{start}) \\ c(\texttt{stop})\end{array}\right.
\]
All events are smaller than a certain maximum \( S_{i,\text{max}} \)
\[
\Rightarrow \exists S_{i,\text{max}} \text{ with } S_{ik} \leq S_{i,\text{max}} \forall k \in \{1, \ldots, N_i\}
\]
\[
\Rightarrow c(\texttt{start}) + c(\texttt{stop}) + \sum_{k=1}^{N_i} S_{ik} + \alpha_i \cdot \sum_{k=1}^{N_i} S_{ik}
\]
\[
\leq 2 \cdot C_{\texttt{start/stop}} + N_i \cdot S_{i,\text{max}} + \alpha_i \cdot N_i \cdot S_{i,\text{max}}
\]
\[
\leq 2 \cdot C_{\texttt{start/stop}} + N_i \cdot S_{i,\text{max}} + D \cdot N_i \cdot S_{i,\text{max}}
\]
\[
\Rightarrow \text{The communication complexity is in}
\[
O(N_i)
\]
\( \square \)
5.3.2 Conductor

Lemma 7 (Initialization and Shutdown). The communication complexity of the Conductor is in

\[ O(m) \]

Proof. For every Monitoring Task the Conductor receives an add_monitor request during initialization and a remove_monitor request during shutdown. Using \( c(\text{start}) \) and \( c(\text{stop}) \) just like in Lemma 1, one gets:

\[ m \cdot (c(\text{start}) + c(\text{stop})) \]

\( c(\text{start}) \) and \( c(\text{stop}) \) are constant

\[ \Rightarrow \exists C_{\text{start/stop}} \in \mathbb{N} : C_{\text{start/stop}} \geq \begin{cases} c(\text{start}) \\ c(\text{stop}) \end{cases} \]

\[ \Rightarrow m \cdot (c(\text{start}) + c(\text{stop})) \leq m \cdot 2 \cdot C_{\text{start/stop}} \]

\( \Rightarrow \) The communication complexity is in

\[ O(m) \]

Lemma 8 (Communication). The communication complexity of the Conductor is in

\[ O(0) \]

Proof. This lemma follows immediately, as the Conductor is not involved in communication anymore.
Chapter 6

Performance Tests

6.1 Large Scale Tests

During the Large Scale Test (LST) programme the lxshare cluster consisting of 700 machines has been allocated for testing of the reimplementaiton presented in this thesis for one night. This chapter describes the cluster topology of lxshare and presents the results of this test.

6.1.1 Network topology of Large Scale Testing Cluster

The general layout of computing nodes and how they are interconnected can be seen in figure 6.1. All computing nodes used for tests are dual CPU machines with at least 1 GHz and 512 MB of memory. For the tests described in this chapter, machines from the vault of building 513 have been used, which are arranged in 9 clusters, each consisting of 42 to 94 machines. Each cluster consists of 4 groups, machines inside each group being interconnected via Fast Ethernet connections and a Fast Ethernet switch, group switches are interconnected via Gigabit links, clusters via Gigabit links and Gigabit routers.

6.1.2 Large Scale Test Aim

The aim of the Event Monitoring scalability tests was to prove performance and scalability of the reimplementaiton presented in this thesis. A bottleneck of the previous implementation, the central Distributor has been removed following the Peer-to-Peer paradigm, which should result in improved scalability on the Monitoring Task side. Another aim of the test was to evaluate the overhead in the Conductor application, induced by measures guaranteeing fault tolerance (like Event Sampler pinging) and efficiency (like sampling rate adaptation). Finally, the tests have been used to evaluate the network transfer overhead generated by CORBA in regard to different event sizes.
6.1 Large Scale Tests

Figure 6.1: Large Scale Test network topology[16]

6.1.3 Large Scale Test Configurations

For the different test aims, several configurations have been created. In configuration A, N Monitoring Tasks have been connected to one Event Sampler, all Monitoring Tasks requesting events matching the same selection criteria. As described in chapter 4.1, in this case Monitoring Tasks are arranged in k-nary trees, parameter k depending on the Conductor configuration. During the tests, different values for k have been used, for configuration A tested parameters include 1, 2, 5 and 10. For this configuration the most interesting value is the number of events per second received by the Monitoring Tasks. In configuration B, a single Event Sampler application and different numbers N of Monitoring Tasks, all requesting events with different selection criteria have been started, resulting in Event Sampler applications running N concurrent event channels, each served by a different thread. For this configuration, the most interesting value is the number of events per second received by the connected Monitoring Tasks, which reflects the rate at which the event channels could ship events. Every test in configuration A and B has been performed using three different profiles, ROD, ROS and EventBuilder profile, using event sizes of 512, 2096 and 500000 4-byte words.

6.1.4 Large Scale Test Approach and Execution

For the execution of tests, test applications from the DAQ/HLT-1 release have been used. Test scripts started a given number of test applications on the testbed.
machines using the "\texttt{ssh -x}" command. The test applications produced local raw result data files, which were copied to a central directory on AFS and parsed by an analysis script, calculating mean values and formatting output data to result files, from where they could be plotted using Excel and GnuPlot. General measurands that have been recorded for all test applications involve CPU time spent, total time spent and events shipped or received per second.

### 6.1.5 Large Scale Test Results

After completion of the tests a bug has been detected concerning CORBA's memory management. This bug was present in the version of the Event Monitoring framework used for the tests and renders all measurand recorded in child Monitoring Tasks for configuration A unusable, as in the case of child Monitoring Task no actual event data but zero-size events have been received. Although this bug has been fixed in current versions, there was no time to repeat tests on the LST or configuration A with the patched version. Regarding this bug, one has to be careful with the interpretation of figure 6.2, which shows the event rate in events per second in the ROD profile against the number of concurrent Monitoring Tasks, all requesting events matching the same selection criteria. Actually the value resulting from the test with a single Monitoring Task

![Figure 6.2: Results of Configuration A](image)

is significant, as only child Monitoring Tasks did receive zero length data, root Monitoring Tasks were not affected by that bug. Using the ROD profile, a single Monitoring Task connected to a Event Sampler application reaches a net data transfer rate of about 30 MBit/s, which is less than one would expect from a 100 MBit connection. Comparing this value to the ROD-profile values recorded in configuration D, which is about 16.9 Mbit/s per Monitoring Task, accumulating to a total data transfer rate of 84.6 Mbit/s in the Event Sampler application this leads to the conclusion, that Monitoring Tasks are the limiting factor, regarding how many \texttt{pushEvent} calls per second they can handle at maximum. A reason for this might be locking mechanisms used in order to synchronize concurrent access to event buffers by the user and the \texttt{pushEvent} calls, but this still has to be investigated in more detail. In a recently updated version of the reimplementation locking scopes have been minimized in order to improve performance. Another reason might be limitations in the CORBA implementation.
used for the Event Monitoring framework. As one can see in figure 6.2 a single Monitoring Task in the EB profile - with less frequent pushEvent calls and much more data carried by each call - receives event data at the maximum rate at which the Event Sampler application can ship them, which is about 85 Mbit/s. Although the bug prevented actual data from being transferred between parent

Figure 6.3: Results of Configuration B

![Graphs showing results of Configuration B](image)

and child Monitoring Tasks, the almost linear scalability in the number of concurrent Monitoring Tasks that can be seen in figure 6.2 is what one would expect from the reimplementation. As no central Conductor is involved in actual data transfer and all the Monitoring Tasks are participating in data distribution, there should be no more bottleneck preventing scalability. In the former implementation, even with zero length data, the rate at which nextEvent calls by the Monitoring Tasks could be answered by the central Distributor would have dropped linearly with an increase in the number of Monitoring Tasks due to the overhead connected with each call (although empty). Figure 6.2 shows that at least with empty nextEvent calls this problem is solved. Figure 6.3 shows the received event rate of Monitoring Tasks in events per second for the ROD and EB profiles in scenario B. In this configuration, Event Sampler applications have to create a different sampling channel for every connected Monitoring Task, transferring data to every single connected Monitoring Task, instead of transferring data to a root Monitoring Task only, letting it spread event data among its children. This does not scale in the number of event channels, but there does not seem to be a proper solution for this problem. Event data actually have to be transferred several times, as they are different for each channel respectively for each Monitoring Task. In the reimplementation, at least this does not influence other Event Sampler applications connected to the same Conductor, so the effect is limited to Monitoring Tasks connecting to that Event Sampler application, rather than having influence on the Event Monitoring performance of a whole partition. For EB profile tests with 5 concurrent event channels, every connected Monitoring Task receives 1.17 events per second, so the overall data rate at which the Event Sampler application ships events is roughly 5 * 1.177 * 500,000 * 4 bytes = 11.224 MB/s = 89.8 MBit/s, which roughly matches the value one would expect from a satisfied 100 MBit link, considering the data overhead generated by RPC marshalling. In order to investigate the bandwidth utilization by the Event Sampler application, figure 6.4 shows the net data rate in MBit/s at which the Event Sampler application ships event to all event channel subscribers, using different event profiles. This is actually
interesting in order to evaluate the overhead generated by the RPC paradigm used in CORBA. Naturally, the overhead is the bigger, the more remote procedure calls have to be performed, as the ratio between payload and marshalled data becomes worse. This is the reason for the increase in the net data transfer rate, which can be clearly seen in figure 6.3. The bigger the event size, the more beneficial it is for the net data transfer rate, resulting in the above mentioned net data transfer rate of 89.8 Mbit/s for the EB profile. Nevertheless with bigger event sizes the net data transfer rate will converge to a value slightly less than 100Mbit/s, because of the data overhead generated by TCP and MAC fragmentation.

Figure 6.4: Accumulated throughput over 5 sampling channels for a single Event Sampler

6.1.6 Summary

As already mentioned, the scalability of Event Sampler with regard to the number of Monitoring Tasks with the same selection criteria was not proved because of the aforementioned bug in the version that has been tested. It should be noted that there seems to be a limitation on the Monitoring Task side, regarding how many pushEvent calls it can handle per time interval. This certainly is an important outcome of the Large Scale Test period and deserves some further investigation as mentioned in chapter 6.1.7. Concerning scenario B, there should be a discussion among users how many event channels per Event Sampler application they will need. If there is a need for many channels per Event Sampler application, there should be some considerations on how to get rid of the bottleneck described above. On the other hand, if having only one or at maximum few channels per Event Sampler application will be the common case, a solution for this might not be required.

6.1.7 Future Tests

Concerning the maximum data transfer rate of 30 Mbit/s at which Monitoring Tasks in the ROD profile seem to be able to handle pushEvent calls, some more investigations on the reason for this shall be made. A dummy Monitoring Task
implementation receiving ROD profile event data at maximum speed, but not putting them to a buffer, not doing any locking should show whether this is a CORBA specific limitation or caused by locking overhead. In order to get results comparable to those captured during this year’s LST period this dummy implementation should be included in the next LST tests. Considering the novelty of the reimplemention, there is still much room for feedback and user suggested improvements. Recently, this feedback lead to the API extension supporting segmented memory mentioned in chapter 4.7. As the version tested during the LST test period did not yet include the implementation of smart pointers and the lovec API, this will certainly be something worth testing in the next test programme. Another point that might be interesting for further testing is the difference between Java and C++ Monitoring Task performance.

6.2 Scalability Comparison

As the bug mentioned in section 6.1.5 rendered test measurands for multiple Monitoring Tasks connected to one Event Sampler application - all sampling the same selection criteria - unusable, this test has been repeated with the current version of the reimplemention not suffering from this bug anymore. Unfortunately for these tests the LST cluster could not be used, so all tests presented in this section have been done using CERN’s regular computing cluster LXPLUS. As these machines are used by many users concurrently, one has to be careful with the interpretation of results. In order to receive any meaningful values at all, several test runs for every configurations have been done, using the mean value of the runs with the best performance as actual result. Apart from that tests have been performed at night, when the expected utilization of LXPLUS was low. For the reimplemention, all test results presented in this section are actually results captured in the leaf Monitoring Tasks. For the tests, all Event Monitoring applications were running on different machines of the lxplus cluster, the machines being interconnected with Fast Ethernet links. As the test aim was to prove scalability in the number of concurrent Monitoring Tasks, a single configuration has been used, connecting N Monitoring Tasks to a single Event Sampler, all Monitoring Tasks sampling events that match the same selection criteria with a size of 30 KB per event. As it is hard to find a number of LXPLUS machines which are not heavily used for a couple of minutes and therefore tests had to be performed by hand, values of N that have been tested are not greater than 5. Test runs have been done with the previous implementation and the reimplemention in order to be able to compare scalability, in each run 10000 events have been transferred between the single Event Sampler and Monitoring Tasks, the maximum node degree of the Monitoring Tree used for the tests was 1, resulting in a simple list of Monitoring Tasks.

6.2.1 Test Results

Figure 6.5 shows the throughput per Monitoring Task comparing the reimplemention and the previous implementation in Mbit per second. Although
having a quite big variation in values due to the utilization of LXPLUS by other users, one can easily see, that the throughput per Monitoring Task in the previous implementation drops by more than 50 % as the number of concurrently running Monitoring Tasks connected to a single event channel increases. In the reimplementation the throughput per Monitoring Task remains remarkably constant when the number of Monitoring Tasks increases. Figure 6.6 shows the accumulated throughput of all concurrently running Monitoring Tasks in Mbit per second. While in the previous implementation this value converges towards a maximum value (which is limited by the bandwidth of the machine running the central Distributor), in the reimplementation one can see a linear increase of the accumulated throughput when the number of concurrent Monitoring Tasks increases, as postulated earlier in chapter 4.1.

6.2.2 Summary

The test results presented above show, that the reimplementation scales in the number of concurrent Monitoring Tasks and rids the Event Monitoring System from the Distributor bottleneck of the previous implementation also when actual data are transferred. It also shows a linear increase in the accumulated throughput of the Event Monitoring System, while the accumulated throughput of all Monitoring Tasks in the previous implementation converges against a value limited by the bandwidth of the central Distributor machine. Regarding the utilization of the LXPLUS cluster, the tests presented above shall be repeated in one of the next LST test periods in order to get results free from side-effects caused by other users.
Figure 6.6: Accumulated throughput of multiple concurrent Monitoring Tasks
Chapter 7

Conclusion and Future Steps

Having compared performance of the previous implementation and the reimplementation of the LHC ATLAS Event Monitoring System done in the course of this thesis both in a theoretical and a practical way, and presented the general architecture of both implementations, it is clear that the reimplementation on the one hand fulfills all the requirements listed in [9], while on the other hand it overcomes the drawbacks mentioned in chapter 3.3. Apart from offering an improved overall speed and improved scalability as presented in chapter 6.2, it offers a configurable trade-off between latency and bandwidth requirements by letting users define the maximum node degree in the Monitoring Tree, it rids users using pull model Event Sampler API from doing any thread management and it offers enhanced fault tolerance behavior. As one can see comparing the theoretical communication complexity analysis done in chapter 5, the complexity in the communication phase in the Conductor application has been reduced to zero, with only constant additional expenses in the Monitoring Tasks.

Although the reimplementation presented in this thesis overcomes several drawbacks of the previous implementation, there still is room for some improvements. One might think of a situation where the root Monitoring Task is running on a machine with low bandwidth. In this situation in the current version of the reimplementation the low bandwidth root Monitoring Task will be a bottleneck for all other Monitoring Tasks sampling the same selection criteria and sampling address. In this case one might think of an implementation where nodes in the Monitoring Tree are automatically arranged according to their bandwidth, small bandwidth Monitoring Tasks becoming leaf, high bandwidth Monitoring Tasks becoming root and internal Monitoring Tasks. This certainly is a solution worth being evaluated for future versions of the Event Monitoring System, although right now it looks like the case of many Monitoring Tasks sampling the same selection criteria and sampling address will not be of much relevance for the final usage scenario.

The reimplementation presented in this thesis has recently been added to CERN CVS, being now included in regular releases of the TDAQ software.
Appendix A

ATLAS Event Monitoring
System Class Reference

A.1  emon::BadAddress Struct Reference

Exception being thrown by emon::select.

#include <Common.h>

Inheritance diagram for emon::BadAddress:

```
emon::CannotInitialize
     ^
emon::BadAddress
```

Public Member Functions

- void raise ()

Public Attributes

- const std::string what_

A.1.1 Detailed Description

Exception being thrown by emon::select.

This exception will be thrown by emon::select in case a wrong sampling address was specified. This usually means, that no EventSampler(p. 106) exists which is responsible for this sampling address.
See also:
   emon::select

Author:
   Ingo Scholtes

The documentation for this struct was generated from the following file:

- Common.h
A.2 emon::BadCriteria Struct Reference

Exception being thrown by emon::select.

```
#include <Common.h>
```

Inheritance diagram for emon::BadCriteria::

```
emon::CannotInitialize
    
emon::BadCriteria
```

**Public Member Functions**

- void raise ()

**Public Attributes**

- const std::string what_

A.2.1 Detailed Description

Exception being thrown by emon::select.

This exception will be thrown by emon::select in case a wrong selection criteria was specified. This usually means, that the **EventSampler** (p.106) associated with the given sampling address does not accept the selection criteria. This exception may be thrown by custom **EventSampler** (p.106) applications, it is however never thrown from within the framework!

**See also:**

- emon::select **PullSampling** (p.124) **PushSampling** (p.128)

**Author:**

Ingo Scholtes

The documentation for this struct was generated from the following file:

- Common.h
A.3  emon::CannotInitialize Struct Reference

Exception being thrown in case Event(p.76) Sampler or Monitoring Task can not initialize.

#include <Common.h>

Inheritance diagram for emon::CannotInitialize::

```
emon::CannotInitialize

emon::BadAddress  emon::BadCriteria  emon::NoResources
```

Public Member Functions

- **CannotInitialize** (const char *what)
- virtual void **raise** ()

Public Attributes

- const std::string **what**

A.3.1  Detailed Description

Exception being thrown in case Event(p.76) Sampler or Monitoring Task can not initialize.

This exception will be thrown by select in case the Monitoring Task can not initialize for some reason. It will also be thrown by the constructor of class EventSampler(p.106) if EventSampler(p.106) can not initialize. This class is the base class for BadAddress(p.61), BadCriteria(p.63) and NoResources(p.123).

See also:
emon::select  EventSampler(p.106)  BadAddress(p.61)  BadCriteria(p.63)  NoResources(p.123)

Author:
Sergei Kolos

The documentation for this struct was generated from the following file:

- Common.h
A.4 emon::Conductor_impl Class Reference

A class representing the Conductor object.

#include "Conductor_impl.h"

Public Member Functions

- **Conductor_impl** (const IPCPartition &p, unsigned int max_children, bool allow_adaptation)
  
  Constructor.

- **~Conductor_impl** ()
  
  Destructor.

- **void throwInitializationException** ()
  
  Throws any emon::CannotInitialize(p.64) exception that might have occurred in the constructor.

Static Public Member Functions

- **bool isExist** (const IPCPartition &p)
  
  Tells whether an Conductor already exists in this partition.

Private Types

- **typedef std::map< std::string, MonitorNode * > EventMonitors**
  
  A STL Map containing references to all root Monitoring Task nodes for all event channels.

- **typedef std::map< std::string, EventMonitoring::EventSampler_var > EventSamplers**
  
  A STL Map containing references to all EventSamplers for all sampling addresses.

Private Member Functions

- **void connect_sampler** (const EventMonitoring::SamplingAddress &sa, EventMonitoring::EventSampler_ptr sampler)
  
  Registers a new EventSampler(p.106).

- **void disconnect_sampler** (const EventMonitoring::SamplingAddress &sa, EventMonitoring::EventSampler_ptr sampler)
Delete an event sampler from the Conductors local EventSampler (p. 106) list.

- void add_monitor (const EventMonitoring::SamplingAddress &sa, const EventMonitoring::SelectionCriteria &sc, EventMonitoring::EventMonitor_ptr mon)
  
  Connect a Monitoring Task to the EventMonitoring System.

- void remove_monitor (const EventMonitoring::SamplingAddress &sa, const EventMonitoring::SelectionCriteria &sc, EventMonitoring::EventMonitor_ptr mon)
  
  Disconnects a Monitoring Task from the EventMonitoring System.

- void adapt_sampling_rate (const EventMonitoring::SamplingAddress &sa, const EventMonitoring::SelectionCriteria &sc, EventMonitoring::EventMonitor_ptr mon, EventMonitoring::Direction direction)
  
  Adjusts the sampling rate of an EventSampler (p. 106).

- void shutdown ()
  
  Shuts down the EventMonitoring Conductor.

- void removeSampler (EventSamplers::iterator it)
  
  Removes an EventSampler (p. 106) from the local list.

- EventSamplers::iterator findSampler (const EventMonitoring::SamplingAddress &sa, std::string &name)
  
  Finds an EventSampler (p. 106) in the samplers_map and returns the appropriate iterator.

Static Private Member Functions

- bool pingLoop (void *param)
  
  Sends a periodic ping to all EventSamplers.

Private Attributes

- IPCAlarm * ping_
  
  IPCAlarm, taking care of a periodical ping to all EventSamplers.

- unsigned int max_children_
  
  The maximum amount of children each Monitoring Task may have.

- bool adaptation_
Whether or not this Conductor allows sampling rate adaptation.

- **EventMonitors monitors**
  A STL map containing string representations of sampling address and selection criteria, associated with a reference to a root MonitorNode (p. 118) object that represents the appropriate root Monitoring Task.

- **ConductorInfoNamed isinfo**
  An IS Info object which contains statistical information about the Conductor.

- **EventSamplers samplers**
  A STL map containing string representations of sampling address and selection criteria, associated with the CORBA object reference to the EventSampler (p. 106) object.

- **OWLMutex mutex**
  A mutual exclusion object for internal use.

- **CannotInitialize *exception**
  A reference to an exception, that might have been thrown in the constructor.

### A.4.1 Detailed Description

A class representing the Conductor object.

An object of type **Conductor_impl** (p. 65) is the central manager of all EventSamplers in a partition. It is not used for communication itself but for connection management, error recovery and encapsulation of data flow. It uses IPCs singletread behavior, as all operations may be serialized

**Author:**
Ingo Scholtes

### A.4.2 Constructor & Destructor Documentation

**A.4.2.1 emon::Conductor_impl::Conductor_impl (const IPCPartition & p, unsigned int max_children, bool allow_adaptation)**

Constructor.

Creates a new **Conductor_impl** (p. 65) object, using the specified parameters. The constructor does some initialization of values in IS server and takes care of all necessary initialization. The maximum node degree in the Monitoring Tree can be specified by setting max_children to the appropriate value, a value of k will result in a k-ary tree. The constructor will also care about reobtaining information about all eventually running EventMonitoring applications.
Parameters:

- \( p \) the IPC Partition to work in
- \( \text{max\_children} \) the maximum amount of children, each Monitoring Task in the Monitoring Tree might have
- \( \text{allow\_adaptation} \) whether to allow sampling rate adaptation or not

A.4.3 Member Function Documentation

A.4.3.1 \texttt{void emon::Conductor\_impl::adapt\_sampling\_rate} \\
(const \texttt{EventMonitoring::SamplingAddress} & \( sa \), \\
const \texttt{EventMonitoring::SelectionCriteria} & \( sc \), \\
\texttt{EventMonitoring::EventMonitor\_ptr monitor}, \\
\texttt{EventMonitoring::Direction direction}) [private]

Adjusts the sampling rate of an \texttt{EventSampler}(p.106).

This method is called by EventMonitors to adjust the speed of a sampling channel to their processing/forwarding capacity. It only accepts requests from root Monitoring Tasks, as only these are directly connected to the EventSamplers. All other Monitoring Tasks do not know, whether they are directly connected to an \texttt{EventSampler}(p.106) or to another Monitoring Task, so they will send requests too, which this method ignores. This method will only forward requests to the \texttt{EventSampler}(p.106) if sampling rate adaptation is allowed in this partition, otherwise it will throw \texttt{NotAllowed} exception. This is an IPC published method.

Parameters:

- \( sa \) the sampling address of the \texttt{EventSampler}(p.106) the Monitoring Task wants to adjust
- \( sc \) the selection criteria of the sampling thread the Monitoring Task wants to adjust
- \( monitor \) a CORBA object reference of the Monitoring Task issuing the request
- \( direction \) the direction of the adaptation request, either \texttt{SpeedUp} or \texttt{SlowDown}

Exceptions:

- \texttt{EventMonitoring::NotAllowed}

See also:

- EventMonitoring::Direction

A.4.3.2 \texttt{void emon::Conductor\_impl::add\_monitor} \\
(const \texttt{EventMonitoring::SamplingAddress} & \( sa \), \\
const \texttt{EventMonitoring::SelectionCriteria} & \( sc \), \\
\texttt{EventMonitoring::EventMonitor\_ptr monitor}) [private]

Connect a Monitoring Task to the EventMonitoring System.
This either sends a new root subscription to the appropriate EventSampler (p. 106) or attaches the calling Monitoring Task to an already existing Monitoring Tree. It will also update some status information about the Event-Monitoring Conductor in the IS server. his is an IPC published method.

Parameters:

- **sa** the sampling address of the EventSampler (p. 106) the Monitoring Task wants to sample events from
- **sc** the selection criteria, that all sampled events shall satisfy
- **monitor** a CORBA object reference to the calling Monitoring Task

A.4.3.3 **void emon::Conductor_impl::connect_sampler**

(const EventMonitoring::SamplingAddress & sa,
 EventMonitoring::EventSampler_ptr sampler) [private]

Registers a new EventSampler (p. 106).
This is called by EventSamplers and will register a new EventSampler (p. 106) in the Conductors local list. This is an IPC published method.

Parameters:

- **sa** the sampling address of the calling EventSampler (p. 106)
- **sampler** a CORBA object reference to the calling EventSampler (p. 106)

A.4.3.4 **void emon::Conductor_impl::disconnect_sampler**

(const EventMonitoring::SamplingAddress & sa,
 EventMonitoring::EventSampler_ptr sampler) [private]

Delete an event sampler from the Conductors local EventSampler (p. 106) list.
This method may be invoked either by EventSamplers themselves or by the thread responsible for pinging all EventSamplers in the Conductor in periodic intervals. In any case it notifies the Conductor about the exit of an EventSampler (p. 106), so it can delete the EventSampler (p. 106) from its local list, delete the Monitoring Tree and tell the root Monitoring Task that the Event (p. 76) Sampler has exited. The root Monitoring Task will then spread the news among all its children. This is an IPC published method.

Parameters:

- **sa** the sampling address of the EventSampler (p. 106) to be deleted
- **sampler** a CORBA object reference to the EventSampler (p. 106) to be deleted

A.4.3.5 **bool emon::Conductor_impl::isExist** (const IPCPartition & p) [static]

Tells whether an Conductor already exists in this partition.
Parameters:
  \( p \) the partition in which to look for Conductor objects

A.4.3.6 bool emon::Conductor_impl::pingLoop (void * \( param \))
[static, private]

Sends a periodic ping to all EventSamplers.
This will start a loop, sending ping requests to all EventSamplers currently
known by this Conductor, to see whether they are still alive or not. If a
ping fails, it will result in a call to removeSampler for the appropriate Event-
Sampler(p.106). As this might cause load in the EventSamplers, as well as
load in the EventMonitoring Conductor, the ping interval should be quite long.
10 seconds is the standard value.

Parameters:
  \( param \) a (void *) casted reference to the instance of Conductor_-
impl(p.65) we are calling this method from

Returns:
  always returns true

A.4.3.7 void emon::Conductor_impl::remove_monitor
(const EventMonitoring::SamplingAddress & \( sa \),
const EventMonitoring::SelectionCriteria & \( sc \),
EventMonitoring::EventMonitor_ptr \( monitor \)) [private]

Disconnects a Monitoring Task from the EventMonitoring System.
This cancels a subscription of a Monitoring Task. This method takes care of
rebuilding the Monitoring Tree (in case there are any Monitoring Tasks left,
connected to this EventSampler(p.106)) or cancellation of the root subscrip-
tion in the EventSampler(p.106). This method might be called by Monitoring
Tasks themselves having determined, that a child has exited or by the Event-
Samplers having noticed the crash of a Monitor Task. This is an IPC published
method.

Parameters:
  \( sa \) the sampling address the monitor was connected to
  \( sc \) the selection criteria of this subscription
  \( monitor \) the CORBA object reference of the Monitoring Task, that shall
  be removed

A.4.3.8 void emon::Conductor_impl::removeSampler
(EventSamplers::iterator \( it \)) [private]

Removes an EventSampler(p.106) from the local list.
This removes an EventSampler(p. 106) from the samplers_ map and notifies the connected root Monitoring Tasks about the exit of the EventSampler(p. 106). This method is used by disconnect_sampler method and the pingLoop

See also:
  samplers_(p. 72) disconnect_sampler(p. 69) pingLoop(p. 70)

A.4.3.9  void emon::Conductor_impl::shutdown () [private]

Shuts down the EventMonitoring Conductor.
This method shuts down the EventMonitoring Conductor by calling stop()

A.4.3.10  void emon::Conductor_impl::throwInitializationException()
  [inline]

Throws any emon::CannotInitialize(p. 64) exception that might have occurred in the constructor.
This is needed, because in case of an exception being thrown in the constructor we want a proper cleanup. Instead of interrupting the constructor, we continue and destroy the object properly in case an exception has occurred.

See also:
  emon::CannotInitialize(p. 64)

A.4.4  Member Data Documentation

A.4.4.1  unsigned int emon::Conductor_impl::max_children_
  [private]

The maximum amount of children each Monitoring Task may have.
The maximum amount of children each Monitoring Task may have great influence on the overall performance of the EventMonitoring system considering bandwidth/latency tradeoff.

A.4.4.2  EventMonitors emon::Conductor_impl::monitors_
  [private]

A STL map containing string representations of sampling address and selection criteria, associated with a reference to a root MonitorNode(p. 118) object that represents the appropriate root Monitoring Task.

See also:
  MonitorNode(p.118)
A.4.4.3 EventSamplers emon::Conductor_impl::samplers
[private]

A STL map containing string representations of sampling address and selection criteria, associated with the CORBA object reference to the EventSampler(p.106) object.

See also:
   EventSampler_impl(p. 107)

The documentation for this class was generated from the following files:

- Conductor_impl.h
- Conductor_impl.cc
A.5 ConductorInfoNamed Class Reference

Contains information about the EventMonitoring Conductor subsystem.
#include <ConductorInfoNamed.h>

Public Member Functions

- ConductorInfoNamed (const IPCPartition &partition, const std::string &name)

Public Attributes

- unsigned long totalMonitors
- unsigned long adaptReceived
- unsigned long adaptAccepted
- unsigned long totalSamplers
- unsigned long activeMonitors
- unsigned long activeSamplers
- unsigned long crashedSamplers
- unsigned long treeType
- bool previousConductorCrashed
- unsigned long crashedConductors

Protected Member Functions

- ConductorInfoNamed (const IPCPartition &partition, const std::string &name, const std::string &type)
- void publishGuts (iostream &out)
- void refreshGuts (iostream &in)

Private Member Functions

- void initialize ()

A.5.1 Detailed Description

Contains information about the EventMonitoring Conductor subsystem.

Author:

IS code generation tool
A.5.2 Member Data Documentation

A.5.2.1 unsigned long ConductorInfoNamed::activeMonitors
number of active Monitoring Tasks connected to this Conductor

A.5.2.2 unsigned long ConductorInfoNamed::activeSamplers
number of active EventSamplers connected to this Conductor

A.5.2.3 unsigned long ConductorInfoNamed::adaptAccepted
total number of adaptation requests forwarded to EventSamplers

A.5.2.4 unsigned long ConductorInfoNamed::adaptReceived
total number of adaptation requests received from Monitoring Tasks

A.5.2.5 unsigned long ConductorInfoNamed::crashedConductors
number of Conductor crashes since start of this IS server

A.5.2.6 unsigned long ConductorInfoNamed::crashedSamplers
total number of crashed EventSamplers

A.5.2.7 bool ConductorInfoNamed::previousConductorCrashed
tells, whether we were started during active connections due to a previous un-
dean exit

A.5.2.8 unsigned long ConductorInfoNamed::totalMonitors
total number of Monitoring Tasks that have been connected to this Conductor

A.5.2.9 unsigned long ConductorInfoNamed::totalSamplers
total number of EventSamplers that have been registered with this Conductor

A.5.2.10 unsigned long ConductorInfoNamed::treeType
type of the Monitoring Tree (maximum node degree per Monitoring Task)
The documentation for this class was generated from the following file:
- ConductorInfoNamed.h
A.6  emon::Event Class Reference

A class representing an event.
#include <Common.h>

Public Member Functions

- const bool isContiguous ()
  
  Returns whether or not the event consists of a single event fragment.

- const unsigned long * data () const
  
  Get the data of all fragments in a single contiguous memory chunk.

- unsigned long size () const
  
  Returns the overall length of this event.

- Event (EventMonitoring::EventFragment *fragments, unsigned long count, bool orphan=true)

  Constructor.

Friends

- class EventMonitor_impl

A.6.1  Detailed Description

A class representing an event.
This class represents an event, consisting of a sequence of event fragments. It inherits the class generated from the IDL declaration of EventMonitoring::Event

Author:
  Ingo Scholtes

A.6.2  Constructor & Destructor Documentation

A.6.2.1  emon::Event::Event (EventMonitoring::EventFragment *fragments, unsigned long count, bool orphan = true) [inlines]

Constructor.

Creates a new event consisting of one or more event fragments. By default the newly created event object takes ownership of the event fragment buffers.
Parameters:

- **fragments** an array of event fragments this event consists of
- **count** number of event fragments
- **orphan** CORBA internal parameter controlling memory ownership, default value is true

### A.6.3 Member Function Documentation

#### A.6.3.1 `const unsigned long emon::Event::data () const [inline]`

Gets the data of all fragments in a single contiguous memory chunk.

This method will return all data of all event fragments in a single contiguous long data array. This method actually simply copies all underlying long data arrays of all fragments to a single array and returns the result.

#### A.6.3.2 `const bool emon::Event::isContiguous () [inline]`

Returns whether or not the event consists of a single event fragment.

This method returns true if the event consists of a single contiguous memory chunk. If this method returns false, the event consists of several event fragments.

#### A.6.3.3 `unsigned long emon::Event::size () const [inline]`

Returns the overall length of this event.

This method will compute the overall length of this event by summing up the length of all event fragments this event consists of.

The documentation for this class was generated from the following file:

- Common.h
A.7  emon::EventChannel Class Reference

Takes care of event sampling for a single event channel.

#include <EventChannel.h>

Public Member Functions

- void pushEvent (unsigned long *event, long event_size)
  * Pushes an event to the connected Monitoring Tree.*

- void pushEvent (iovec *event, long count)
  * Pushes an event to the connected Monitoring Tree.*

Protected Member Functions

- EventChannel (const IPCPartition &partition, const EventMonitoring::SamplingAddress &address, const EventMonitoring::SelectionCriteria &criteria, EventMonitoring::EventMonitor_ptr monitor, EventSampler_impl *parent)
  * Constructor for push model.

- EventChannel (const IPCPartition &partition, const EventMonitoring::SamplingAddress &address, const EventMonitoring::SelectionCriteria &criteria, EventMonitoring::EventMonitor_ptr monitor, EventSampler_impl *parent, PullSampling *sampler)
  * Constructor for pull model.

- virtual ~EventChannel ()
  * Destructor.

- void destroy (bool wait_for_completion)
  * This will remove the IPC publication of this event channel.

- bool failed ()
  * Tells whether the last pushing of an event to the Monitoring Task failed.

- void setActivity (PushSampling *activity)
  * Sets the PushSampling(p. 128) object for this EventChannel(p. 78).

- void replaceMonitor (EventMonitoring::EventMonitor_ptr monitor)
  * Replaces the root Monitoring Task of this EventChannel(p. 78) with the given reference.

- const EventMonitoring::SamplingAddress & address () const
Return the sampling address associated with the parent EventSampler (p. 106).

- const EventMonitoring::SelectionCriteria & criteria () const
  Returns the selection criteria associated with this EventChannel (p. 78).

- EventMonitoring::EventMonitor_ptr monitor () const
  Returns the root Monitoring Task connected to this EventChannel (p. 78).

- void speedUp ()
  Speeds up the sampling process by decreasing the delay.

- void slowDown ()
  Slows down the sampling process by increasing the delay.

- void wait ()
  Wait for the desired delay.

- double delay ()
  Returns the current sampling delay.

Private Member Functions

- EventChannel (const EventChannel &)
  Copy constructor.

- EventChannel & operator= (const EventChannel &)
  Definition of assignment operator.

Static Private Member Functions

- bool pingLoop (void *param)
  Send a periodic ping to all EventSamplers.

Private Attributes

- IPCAlarm * ping_
  IPCAlarm, taking care of a periodical ping to all EventSamplers.

- PushSampling * activity_
  The push sampling object.
• IPC Partition \texttt{partition_}
  
  The IPC Partition to work in.

• Event Monitoring::SamplingAddress \texttt{address_}
  
  The sampling address of this \texttt{EventSampler} (p. 106).

• Event Monitoring::SelectionCriteria \texttt{criteria_}
  
  The selection criteria of this \texttt{EventChannel} (p. 78).

• Event Monitoring::EventMonitor_var \texttt{monitor_}
  
  A CORBA object reference to the root Monitoring Task of this \texttt{EventChannel} (p. 78).

• OWLCondition \texttt{condition_}
  
  A condition variable used for thread suspension.

• \texttt{EventSampler_impl * parent_}
  
  A reference to the parent \texttt{EventSampler} (p. 106).

• bool \texttt{failed_}
  
  A flag telling whether the last call to \texttt{pushEvent} did fail.

• OWL Mutex \texttt{mutex_}
  
  A mutual exclusion object for internal use.

• double \texttt{delay_}
  
  A double specifying delay for sampling rate adaptation.

• \texttt{PullSamplingThread * pull_thread_}
  
  A thread that invokes \texttt{sampleEvent} in a loop until stopped in case pull model constructor has been used.

• OWL Timer \texttt{timer_}
  
  A timer that will be used to determine the time when the last call to \texttt{pushEvent} appeared.

Friends

• class \texttt{EventSampler_impl}

A.7.1 Detailed Description

Takes care of event sampling for a single event channel.
A.7.2 Constructor & Destructor Documentation

A.7.2.1  eMon::EventChannel::EventChannel (const IPCPartition & partition, const EventMonitoring::SamplingAddress & address, const EventMonitoring::SelectionCriteria & criteria, EventMonitoring::EventMonitor_ptr monitor, EventSampler_impl * parent) [protected]

Constructor for push model.

This will create a new event channel using the push model API. Users may use this event channel object in order to push events to Monitoring Tasks. The EventChannel(p.78) will be automatically created by the parent EventSampler(p.106)

Parameters:

  partition the IPC partition of the EventMonitoring System
  address the sampling address of the EventSampler(p.106)
  criteria the selection criteria of this event channel
  monitor a CORBA object reference to the root Monitoring Task
  parent CORBA object reference to the parent EventSampler(p.106), which has created the EventChannel(p.78)

A.7.2.2 eMon::EventChannel::EventChannel (const IPCPartition & partition, const EventMonitoring::SamplingAddress & address, const EventMonitoring::SelectionCriteria & criteria, EventMonitoring::EventMonitor_ptr monitor, EventSampler_impl * parent, PullSampling * sampler) [protected]

Constructor for pull model.

This will create a new EventChannel(p.78) using the pull model API. This will automatically start a new pull sampling thread, using the users custom PullSampling(p.124) object in order to sample events from The EventChannel(p.78) will be automatically created by the parent EventSampler(p.106)

Parameters:

  partition the IPC partition of the Event(p.76) Monitoring System
  address the sampling address of the EventSampler(p.106)
  criteria the selection criteria of this EventChannel(p.78)
  monitor a CORBA object reference to the root Monitoring Task
  parent CORBA object reference to the parent EventSampler(p.106), which has created the EventChannel(p.78)
  sampler the users PullSampling(p.124) object that will be used by the pull sampling thread in order sample events from
A.7.2.3 emon::EventChannel::~EventChannel () [protected, virtual]

Destructor.
This destructor will automatically stop the push sampling thread and the ping loop.

A.7.3 Member Function Documentation

A.7.3.1 const EventMonitoring::SamplingAddress&
emon::EventChannel::address () const [inline, protected]

Return the sampling address associated with the parent EventSampler(p.106).

Returns:
the sampling address associated with the parent EventSampler(p.106)

A.7.3.2 double emon::EventChannel::delay () [inline, protected]

Returns the current sampling delay.

Returns:
the current delay the sampling thread will wait for in every pushEvent call

A.7.3.3 bool emon::EventChannel::failed () [inline, protected]

Tells whether the last pushing of an event to the Monitoring Task failed.

Returns:
whether or not the last pushEvent call failed

A.7.3.4 EventMonitoring::EventMonitor_ptr
emon::EventChannel::monitor () const [inline, protected]

Returns the root Monitoring Task connected to this EventChannel(p.78).

Returns:
a CORBA object reference pointing to the root Monitoring Task currently connected to this EventChannel(p. 78)
A.7.3.5  bool emon::EventChannel::pingLoop (void * param)  
[static, private]  

Send a periodic ping to all EventSamplers.  
This will start a loop, sending ping requests to all EventSamplers currently 
connected to this conductor, to see whether they are still alive or not. If a ping 
fails, it will result in a call to disconnectSampler for the appropriate Event- 
Sampler(p.106). As this might cause load in the EventSamplers, as well as 
load in the Conductor, the ping interval should be long enough. 10 seconds is 
the standard value.  

Parameters:  
param a (void *) casted reference to the instance of Conductor - 
impl(p.63) we are calling this method from  

A.7.3.6  void emon::EventChannel::pushEvent (iovec * event, long 
count)  

Pushes an event to the connected Monitoring Tree.  
This method pushes an event to the connected Monitoring Tree, guaranteeing 
that event memory may be freed as soon as this function returns. This function 
will block until the event has been delivered. There is no guarantee that the 
event actually will be received in any Monitoring Task. This method shall be 
used for segmented memory events  

Parameters:  
event an iovec with event fragments to push to the Monitoring Tasks 

count the number of event fragments in the iovec event  

A.7.3.7  void emon::EventChannel::pushEvent (unsigned long * 
 event, long event_size)  

Pushes an event to the connected Monitoring Tree.  
This method pushes an event to the connected Monitoring Tree, guaranteeing 
that event memory may be freed as soon as this function returns. This function 
will block until the event has been delivered. There is no guarantee that the 
event actually will be received in any Monitoring Task. This method shall be 
used for contiguous memory events  

Parameters:  

event the event to push to the Monitoring Tasks  

event_size the size of the event to push
A.7.3.8 void emon::EventChannel::replaceMonitor
       (EventMonitoring::EventMonitor_ptr monitor)
       [protected]

Replaces the root Monitoring Task of this EventChannel (p. 78) with the given
reference.
This method will replace a Monitoring Task and reset the failed_ variable for
this EventChannel (p. 78)

Parameters:
   monitor the new CORBA object reference, that shall be installed as root
     Monitoring Task

A.7.3.9 void emon::EventChannel::speedUp () [protected]

Speeds up the sampling process by decreasing the delay.
Decreases this EventChannel’s delay if possible. If not it will throw Event-
Monitoring::MaximumReached

Exceptions:
   EventMonitoring::MaximumReached

A.7.3.10 void emon::EventChannel::wait () [protected]

Wait for the desired delay.
This method will be called automatically by pushEvent
The documentation for this class was generated from the following files:

- EventChannel.h
- EventChannel.cc
A.8  emon::EventChannel::PullSamplingThread
Class Reference

Nested class PullSamplingThread (p. 85).

Public Member Functions

- `PullSamplingThread (PullSampling *sampler, EventChannel *channel)`
  
  Constructor.

- `~PullSamplingThread ()`
  
  Destructor.

- `void maybeSuspend ()`
  
  Suspends the thread in case of an error.

- `void resume ()`
  
  Resumes sampling operation in case the thread has been suspended.

- `void start ()`
  
  Starts the sampling thread.

- `void stop ()`
  
  Stops the sampling thread.

Private Member Functions

- `void *run_undetached (void *)`
  
  Method containing the main loop of the pull sampling thread.

Private Attributes

- `PullSampling *sampler_`
  
  A reference to the PullSampling (p. 124) object which will be used to sampleEvents from.

- `OWL::Mutex mutex_`
  
  A mutual exclusion object for internal use.

- `OWL::Condition condition_`
  
  A condition variable used for thread suspension.
• `bool terminated_`  
  A flag telling whether the thread shall be terminated.

• `EventChannel * channel_`  
  A reference to the underlying event channel which will be used to push events to.

### A.8.1 Detailed Description

Nested class `PullSamplingThread` (p. 85).

This nested class representing a sampling thread, which will pull events from the user by invoking `sampleEvent` of the user's custom `PullSampling` (p. 124) object in regular intervals. These intervals may vary due to sampling rate adaptation.

**See also:**
- `PullSampling` (p. 124)

### A.8.2 Constructor & Destructor Documentation

#### A.8.2.1 `emon::EventChannel::PullSamplingThread::PullSamplingThread`  
(PullSampling * `sampler`, EventChannel * `channel`)  
[inline]

Constructor.

**Parameters:**
- `sampler` PullSampling (p. 124) object needed to pull events from
- `channel` the event channel used to push events

### A.8.3 Member Function Documentation

#### A.8.3.1 `void emon::EventChannel::PullSamplingThread::maybeSuspend()`  

Suspends the thread in case of an error.

This method is called after every call to `PullSampling::sampleEvent` method. It will take care that the pull sampling thread is suspended in case a `pushEvent` call failed or the channel was destroyed.

#### A.8.3.2 `void * emon::EventChannel::PullSamplingThread::run_`  
(undetached (void *))  
[private]

Method containing the main loop of the pull sampling thread.

In this method `PullSampling::sampleEvent`, `maybeSuspend` and `sleep` will be called in a loop until the sampling thread will be destroyed.
A.8.3.3 void emon:EventChannel::PullSamplingThread::start ()

Starts the sampling thread.
This method will start an undetached sampling thread

A.8.3.4 void emon:EventChannel::PullSamplingThread::stop ()

Stops the sampling thread.
This method will stop the sampling thread by setting the terminated_ variable to true. It will also wake up the thread in case it has been suspended so it can cleanly exit.

The documentation for this class was generated from the following files:

- EventChannel.h
- EventChannel.cc
A.9  emon::EventChannel::RecoveryThread

Class Reference

Nested class RecoveryThread (p. 88).

Public Member Functions

- **RecoveryThread** (const IPCPartition &partition, const EventMonitoring::SamplingAddress &address, const EventMonitoring::SelectionCriteria &criteria, EventMonitoring::EventMonitor_ptr monitor, POA_EventMonitoring::EventSampler *parent)
  
  Constructor.

Private Member Functions

- void **run** (void *)
  
  This method will be invoked in a separate thread.

Private Attributes

- IPCPartition **partition**
  
  The IPC Partition to search for the Conductor.

- EventMonitoring::SamplingAddress **address**
  
  The sampling address of this EventSampler (p. 106).

- EventMonitoring::SelectionCriteria **criteria**
  
  The selection criteria of this event channel.

- EventMonitoring::EventMonitor_var **monitor**
  
  A CORBA object reference to the Monitoring Task that shall be removed.

- POA_EventMonitoring::EventSampler * **parent**
  
  A CORBA object reference to the EventSampler (p. 106) which initiated this thread.

A.9.1  Detailed Description

Nested class RecoveryThread (p. 88).

Author:

Sergei Kolos
Ingo Scholtes
The only purpose of this class representing a recovery thread, is to call remove_monitor in the Conductor in case of a failed push_event call. Because of deadlock considerations, this has to be decoupled into a separate thread. The function remove_monitor in Conductor will call back replace_monitor in the sampling application, this would lead to a deadlock if done in the same thread!

A.9.2 Constructor & Destructor Documentation

A.9.2.1 emon::EventChannel::RecoveryThread::RecoveryThread
(const IPCPartition & partition, const
EventMonitoring::SamplingAddress & address, const
EventMonitoring::SelectionCriteria & criteria,
EventMonitoring::EventMonitor_ptr monitor,
POA_EventMonitoring::EventSampler * parent)

Constructor.

This constructor will create a new recovery thread

Parameters:

partition the IPC partition to search for the Conductor
address the sampling address of this Event(p.76) Sampler
criteria the selection criteria of this event channel
monitor a CORBA object reference to the Monitoring Task that shall be removed
parent a CORBA object reference to the Event(p.76) Sampler which has initiated this thread

The documentation for this class was generated from the following files:

- EventChannel.h
- EventChannel.cc
A.10 emon::EventFragment Class Reference

A class representing an event fragment.
#include <Common.h>

Public Member Functions

- const unsigned long *data () const
  returns the underlying long data array of this event fragment

- unsigned long size () const
  returns the length of the underlying long data array

Private Member Functions

- EventFragment (unsigned long *event, unsigned long size, bool orphan=true)
  Constructor.

Friends

- class EventMonitor_impl

A.10.1 Detailed Description

A class representing an event fragment.

This class represents an event fragment, consisting of an array of long values. It inherits the class generated from the IDL declaration of EventMonitoring::EventFragment

Author:
Ingo Scholtes

A.10.2 Constructor & Destructor Documentation

A.10.2.1 emon::EventFragment::EventFragment (unsigned long *event, unsigned long size, bool orphan = true) [inline, private]

Constructor.

This will call the constructor of the underlying CORBA sequence
Parameters:
- \textit{event} the event data of this event
- \textit{size} the size of the data
- \textit{orphan} optional value for CORBA internal use

The documentation for this class was generated from the following file:

- Common.h
A.11  emon::EventIterator Class Reference

An iterator object, that may be used to retrieve events.
#include <EventIterator.h>

Public Member Functions

- **EventIterator** (emon::EventMonitor_impl *mon)
  
  Constructor.

- **~EventIterator** ()
  
  Destructor.

- smart_event_ptr **nextEvent** (unsigned long timeout_ms=0)
  
  Retrieves the next event (synchronously).

- smart_event_ptr **tryNextEvent** ()
  
  Retrieves the next event (asynchronously).

- unsigned int **availableEvents** ()
  
  Return the number of events which is currently available in the buffer.

- bool **eventsDropped** ()
  
  Returns whether or not events have been dropped.

Private Attributes

- **emon::EventMonitor_impl *monitor_**
  
  The EventMonitor_impl(p. 95) object, which will be used to receive the events.

A.11.1  Detailed Description

An iterator object, that may be used to retrieve events.

An object of this type is returned by the select method in the emon namespace and can be used to iterate through the events either in asynchronous or synchronous mode

See also:
  emon::select

Author:
  Ingo Scholtes
A.11.2 Constructor & Destructor Documentation

A.11.2.1 emon::EventIterator::EventIterator
(emon::EventMonitor_impl * mon) [inline]

Constructor.

Parameters:

mon the Monitoring Task used for receiving events

A.11.3 Member Function Documentation

A.11.3.1 unsigned int emon::EventIterator::availableEvents () [inline]

Return the number of events which is currently available in the buffer.

Returns the number of events currently buffered in the event buffer of the Monitoring Task

Returns:

the number of events currently buffered

A.11.3.2 bool emon::EventIterator::eventsDropped () [inline]

Returns whether or not events have been dropped.

Returns whether events have been dropped my the Monitoring Task, because of a full event buffer. Users may try to minimize the number of dropped events by optimizing the speed at which they process events in the Monitoring Task, and by maximizing the maximum size of the event buffer. Note: Even when eventsDropped returns false, events might have been dropped by other parts of the emon service

Returns:

whether or not events have been dropped

A.11.3.3 smart_event_ptr emon::EventIterator::nextEvent
(unsigned long timeout_ms = 0) [inline]

Retrieves the next event (synchronously).

Retrieves the next event synchronously from the buffer, blocks until event is available or timeout is exceeded. If timeout is exceeded and no event could be retrieved, this function will throw NoMoreEvents(p.122). If no timeout is specified, this function will never throw NoMoreEvents(p.122)

Parameters:

timeout_ms milliseconds to wait for an event (0 (default) means wait infinitely)
Returns:
   a smart pointer to an event

Exceptions:
   *emon::NoMoreEvents*

A.11.3.4 smart_event_ptr emon::EventIterator::tryNextEvent ()
   [inline]

Retrieves the next event (asynchronously).

Retrieves the next event asynchronously from the buffer. Throws a *NoMoreEvents* (p. 122) exception if no event is available.

Returns:
   a smart pointer to an event

Exceptions:
   *emon::NoMoreEvents*

The documentation for this class was generated from the following file:

- EventIterator.h
A.12  emon::EventMonitor_impl Class Reference

Main class for the Monitoring Task application.

#include <EventMonitor_impl.h>

Public Member Functions

- `EventMonitor_impl` (const IPCPartition &partition, const SamplingAddress &address, const SelectionCriteria &criteria, unsigned int buffer_limit)
  Constructor.

- `~EventMonitor_impl` ()
  Standard destructor.

- `smart_event_ptr nextEvent` (bool async, unsigned long timeout=0)
  Retrieves the first event from the buffer and returns it to the user for processing.

- `unsigned int bufferSize` ()
  Returns the number of events currently being buffered.

- `bool eventsDropped` ()
  Returns whether or not events have been dropped by this Monitoring Task.

- `void throwInitializationException` ()
  Throws any potentially cached exceptions.

- `void ping` ()
  Used for pinging of root Monitoring Task by the EventSampler(p. 106).

- `void destroy` ()
  Calls stop() (p. 102).

Private Member Functions

- `void add_child` (EventMonitor::EventMonitor_ptr new_child)
  Adds a child to the local children list.

- `void remove_child` (EventMonitor::EventMonitor_ptr child)
  Removes one of the children from the local children list.

- `void sampler_exit` ()
  Notifies the Monitoring Task, that the EventSampler (p. 106) has exited for some reason.
• void **push_event** (const EventMonitoring::Event &event)  
  Pushes an event to the local buffer.

• void **get_children** (EventMonitoring::MonitorList &out children)  
  Gets a sequence containing the children currently connected to this Monitoring Task.

• void **retry_adapt** ()  
  Tells a Monitoring Task that he may now be allowed to send sampling rate adaptation requests.

• void **adaptSamplingRate** ()  
  Send an adaptation request to the Conductor.

• void **start** ()  
  Send a connection request to the EventMonitoring Conductor.

• void **stop** ()  
  Stops receiving events and logs off from EventSampler (p. 106).

### Static Private Member Functions

• bool **publishLoop** (void *param)  
  A simple loop that periodically updates information in IS.

### Private Attributes

• IPCPartition **partition**  
  The partition we work in.

• unsigned int **buffer_limit**  
  The maximum number of events buffer may hold.

• **SamplingAddress** **address**  
  The sampling address, this Monitoring Task is connected to.

• **SelectionCriteria** **criteria**  
  The selection criteria, sampled events should satisfy.

• **MonitorInfoNamed** *info*  
  A pointer to the associated IS info object.

• bool **sampler_gone**
Tells whether or not the connected EventSampler (p. 106) is gone.

- **bool event_dropped**
  Tells whether or not this Monitoring Task has dropped an event.

- **EventMonitor.children**
  A list containing our children Monitoring Tasks.

- **std::list<smart_event_ptr> buffer**
  A buffer containing events that have been received, but not yet processed.

- **bool speedup_possible**
  Tells whether the EventSampler (p. 106) is capable of speeding up sampling.

- **bool never_ask_adaptation**
  Tells not to ask for speedup or slowdown again.

- **OWLMutex children_mutex**
  A mutual exclusion object for synchronization of access to the children list.

- **OWLMutex publish_mutex**
  A mutual exclusion object for synchronization of access to the IS object.

- **OWLMutex evt_mutex**
  A mutual exclusion object for synchronization of access to the event buffer.

- **FwdThread * fwdthread**
  A reference to the forwarding thread object.

- **IPCAAlarm * publishing**
  Alarm, which will be used for publishing IS information.

- **OWLMutex evt_cond_mutex**
  A mutual exclusion object for synchronization of access to evt_condition condition variable.

- **OWLCondition evt_condition**
  A condition variable telling whether another event is ready to be pulled from the evt_buffer.

- **CannotInitialize * exception**
  An exception that has been cached during object construction.

- **EventMonitoring::EventMonitor_var self**
  A CORBA object reference to this object.

- **unsigned long last_buffer_size**
Last time's buffer size, necessary to calculate sampling rate adaptation.

- bool exiting_
  A flag telling whether or not this Monitoring Task is about to shutdown.

A.12.1 Detailed Description

Main class for the Monitoring Task application.

Author:
Ingo Scholtes

This is the main class for the Monitoring Task application, implementing functionality that has been defined in the IDL declaration of EventMonitoring::EventMonitor. It uses multi-threaded behavior so the code has to be thread aware. The main objective of this class is receiving and buffering of events, providing functions to deliver the to the user and forwarding of events to children Monitoring Tasks.

See also:
EventMonitoring::EventMonitor

A.12.2 Constructor & Destructor Documentation

A.12.2.1 emon::EventMonitor_impl::EventMonitor_impl (const IPCPartition & partition, const SamplingAddress & address, const SelectionCriteria & criteria, unsigned int buffer_limit)

Constructor.
This creates an EventMonitor_impl(p.95) object and does all necessary initialization. Any exception occurring in the course of initialization will be caught and cached, being thrown later by throwInitializationException

Parameters:
  partition the IPCPartition object, this Monitoring Task should use
  address the sampling address of the EventSampler(p.106) we want to connect to
  criteria the selection criteria the sampled events should satisfy
  buffer_limit the maximum amount of events to buffer in the event buffer

See also:
throwInitializationException(p.95) emon::CannotInitialize(p.64)
A.12.2.2 \texttt{emon::EventMonitor\_impl::\~EventMonitor\_impl ()}

Standard destructor.

Deletes any cached exception

A.12.3 Member Function Documentation

A.12.3.1 \texttt{void emon::EventMonitor\_impl::adaptSamplingRate () [private]}

Send an adaptation request to the Conductor.

This method will determine whether it is necessary to change the sampling rate in the sampling application and will send the appropriate request to the Conductor. The need for adaptation is determined using the current buffer fill rate and the fill rate change.

A.12.3.2 \texttt{void emon::EventMonitor\_impl::add\_child (EventMonitoring::EventMonitor\_ptr new\_child) [private]}

Adds a child to the local children list.

This Method adds a new child to our local children list. This is an IPC published method.

\textbf{Parameters:}

\texttt{new\_child} a CORBA-reference to the child Monitoring Task

A.12.3.3 \texttt{void emon::EventMonitor\_impl::destroy ()}

Calls \texttt{stop()}(p. 102).

\textbf{See also:}

\texttt{stop()}(p. 102)

A.12.3.4 \texttt{void emon::EventMonitor\_impl::get\_children (EventMonitoring::Monitor\_List\_out children) [private]}

Gets a sequence containing the children currently connected to this Monitoring Task.

This method is invoked by the EventMonitoring Conductor in order to reconstruct the Monitoring Trees after a Conductor crash. After having retrieved the root Monitoring Tasks from the EventSamplers, it will call get\_children for the root Monitoring Tasks and its subtrees, diving into it in DFS manner. This is an IPC published method.
Parameters:
  \textit{children} out parameter containing a sequence of MonitorList structs

See also:
  MonitorList EventSampler_impl(p.107)

A.12.3.5 \texttt{smart\_event\_ptr emon::EventMonitor\_impl::nextEvent}
  (bool \texttt{async}, unsigned long \texttt{timeout = 0})

Retrieves the first event from the buffer and returns it to the user for processing.
This method will return the first event available in the event buffer. Note: The behaviour of this function depends on whether or not the asynchronous flag
is set. If asynchronous event retrieval is used, a call to \texttt{nextEvent} will throw\texttt{emon::NoMoreEvents}(p.122) when there is no event available. In case of syn-
chronous event retrieval, it will block until an event is available or the specified
timeout is exceeded. If the \texttt{EventSampler}(p.106) crashes/stop\s in the middle of event sampling, this method will throw \texttt{emon::SamplerStopped}(p.134)

Returns:
  a smart pointer to an event

Exceptions:
  \texttt{SamplerStopped}
  \texttt{NoMoreEvents}

A.12.3.6 \texttt{void emon::EventMonitor\_impl::push\_event (const
  EventMonitoring::Event & \texttt{event}) [private]}

Pushes an event to the local buffer.
This method is invoked by the event channel, a new thread is automatically cre-
dated for each invocation of this method (POAs multithreaded behavior) Events
will be stored in a local buffer, where they are waiting for being processed (by
the user) and for being forwarded to children Monitoring Tasks This is an IPC
published method.

Parameters:
  \texttt{event} event to push to the buffer

See also:
  \texttt{Event}(p.76)

A.12.3.7 \texttt{void emon::EventMonitor\_impl::remove\_child}
  (EventMonitoring::EventMonitor\_ptr \texttt{child}) [private]

Removes one of the children from the local children list.
By calling this method, the Conductor informs Monitoring Tasks that one of the children has exited, and that it can stop forwarding events to it. This method might be called "from within" \texttt{FwdThread}(p. 103), so we need to be careful in order to avoid deadlocks This is an IPC published method.

**Parameters:**

\texttt{child} a CORBA reference to the child that shall be removed

\texttt{A.12.3.8 \hspace{1em} void emon::EventMonitor_impl::retry\_adapt ()}

[private]

Tells a Monitoring Task that he may now be allowed to send sampling rate adaptation requests.

Notifies this Monitoring Task, that he is root Monitoring Task now. He may now retry to send adaptSamplingRate requests This is an IPC published method.

\texttt{A.12.3.9 \hspace{1em} void emon::EventMonitor_impl::sampler\_exit ()}

[private]

Notifies the Monitoring Task, that the \texttt{EventSampler}(p. 106) has exited for some reason.

This will care about child Monitoring Tasks, when the \texttt{EventSampler}(p. 106) this Monitoring Task is connected to has exited. It will invoke sampler\_exit of every child Monitoring Task and it will cause a \texttt{SamplerStopped}(p. 134) exception in the next invocation of nextEvent to be thrown. This is an IPC published method.

See also:

\texttt{emon::SamplerStopped(p. 134)}

\texttt{A.12.3.10 \hspace{1em} void emon::EventMonitor_impl::start ()} [private]

Send a connection request to the EventMonitoring Conductor.

This method will try to connect to an \texttt{Event}(p. 76) Sampler, by sending a connection request to the Conductor, which will either connect this Monitoring Task to an existing Monitoring Tree or install this monitor as new root Monitoring Task of a new \texttt{EventChannel}(p. 78) and start the sampling thread in the \texttt{EventSampler}(p. 106). It is automatically invoked upon object construction. This function may throw \texttt{emon::CannotInitialize}(p. 64), if connection fails for some reason.

See also:

\texttt{emon::CannotInitialize(p. 64)}
A.12.3.11 void emon::EventMonitor_impl::stop () [private]

Stops receiving events and logs off from EventSampler (p. 106).
A call to this method, will tell the Conductor that we want to disconnect from the EventSampler (p. 106)

A.12.4 Member Data Documentation

A.12.4.1 std::list<smart_event_ptr> emon::EventMonitor_impl::buffer_ [private]

A buffer containing events that have been received, but not yet processed.

See also:
   buffer_limit

The documentation for this class was generated from the following files:

- EventMonitor_impl.h
- EventMonitor_impl.cc
A.13  emon::EventMonitor_impl::FwdThread

Class Reference

Inner class representing the forwarding thread.
#include <EventMonitor_impl.h>

Public Member Functions

- void stop ()
  This method will stop the thread.

- void resume ()
  This method will resume the thread.

- void push_event (smart_event_ptr)
  This method will push a new event to the local forwarding buffer.

- void add_child (EventMonitor::EventMonitor_ptr new_child)
  This method will add a new child to the local children list.

- void remove_child (EventMonitor::EventMonitor_ptr child)
  This method will remove a child from the local children list.

- FwdThread (EventMonitor_impl *monitor, unsigned int buffer_limit)
  Constructor.

- ~FwdThread ()
  Destructor.

Private Member Functions

- void run (void *)
  This method will run in a separate thread and will do event forwarding in a
    loop.

Private Attributes

- EventMonitor_impl * monitor_
  A reference to the Monitoring Task we belong to.

- volatile bool started_
Tells whether the thread has been started.

- volatile bool exit_
  Tells whether or not to exit the thread.

- OWLMutex child_mutex_
  A mutual exclusion object for synchronization of access to the local children list.

- OWLMutex evt_mutex_
  A mutual exclusion object for synchronization of access to the local event condition.

- OWLMutex fwd_cond_mutex_
  A mutual exclusion object for synchronization of access to the fwd_condition variable.

- OWLCondition fwd_condition_
  A condition variable used for waiting for new events that can be forwarded.

- std::list< smart_event_ptr > fwd_buffer_
  A buffer containing smart pointers to events that have been received, but not yet forwarded.

- EventMonitors children_
  A local children list containing CORBA object references to Monitoring Tasks.

- unsigned int buffer_limit_
  The maximum number of events to keep in the forwarding buffer.

A.13.1 Detailed Description

Inner class representing the forwarding thread.

An instance of this class will be created by the constructor and started when the first child will be added. This thread starts a loop, getting the first event from fwdbuffer_ and sending it to all children. It will also take care of calling the Conductor in case any children crash or exit.

Author:
  Ingo Scholtes
A.13.2 Member Data Documentation

A.13.2.1 volatile bool emon::EventMonitor_impl::FwdThread::exit_ [private]

Tells whether or not to exit the thread.
This flag will be manipulated by several threads, so it needs to be volatile

A.13.2.2 volatile bool emon::EventMonitor_impl::FwdThread::started_ [private]

Tells whether the thread has been started.
This flag will be manipulated by several threads, so it needs to be volatile
The documentation for this class was generated from the following files:

- EventMonitor_impl.h
- EventMonitor_impl.cc
A.14  emon::EventSampler Class Reference

A wrapper class for the user’s SamplingFactory implementations.
#include <EventSampler.h>

Public Member Functions

- EventSampler (const IPCPartition &partition, const SamplingAddress &address, PushSamplingFactory *factory, unsigned long max_channels=100)

  Creates a new push model EventSampler(p. 106).

- EventSampler (const IPCPartition &partition, const SamplingAddress &address, PullSamplingFactory *factory, unsigned long max_channels=100)

  Creates a new pull model EventSampler(p. 106).

- ~EventSampler ()

  Finalize the EventSampler(p. 106).

- void wait ()

  Blocks the current thread until the stop method is called.

- void stop ()

  Stops the event sampling process.

Private Attributes

- EventSampler_impl * sampler_

  An object reference to the main EventSampler_impl(p. 107) object.

A.14.1  Detailed Description

A wrapper class for the user’s SamplingFactory implementations.
The documentation for this class was generated from the following file:

- EventSampler.h
A.15  emon::EventSampler_impl Class Reference

Main class for the EventSampler(p.106) application.
#include <EventSampler_impl.h>

Public Member Functions

- EventSampler_impl (const IPCPartition &partition, const SamplingAddress &address, PullSamplingFactory *factory, unsigned long max_criteria=100)
  Constructor for pull model.

- EventSampler_impl (const IPCPartition &partition, const SamplingAddress &address, PushSamplingFactory *factory, unsigned long max_criteria=100)
  Constructor for push model.

- ~EventSampler_impl ()
  Destructor.

- void throwInitializationException ()
  Throws any potentially cached exception.

- void destroy ()
  Stops all sampling threads, unregisters the EventSampler(p.106) and destroys the IPCObject.

Public Attributes

- SamplerInfoNamed info
  A pointer to the associated IS info object.

Private Types

- typedef std::map<std::string, EventChannel *> SamplingChannels

Private Member Functions

- void constructor ()
- void ping ()
  Used as a proof that this EventSampler(p.106) is alive.
• **void** `connect_monitor` (const EventMonitoring::SamplingAddress &address, const EventMonitoring::SelectionCriteria &criteria, Event-Monitoring::EventMonitor_ptr monitor)

  Connects a Monitoring Task to this EventSampler(p. 106).

• **void** `replace_monitor` (const EventMonitoring::SamplingAddress &address, const EventMonitoring::SelectionCriteria &criteria, Event-Monitoring::EventMonitor_ptr monitor)

  Replaces an existing root Monitoring Task with the new one.

• **void** `disconnect_monitor` (const EventMonitoring::SamplingAddress &address, const EventMonitoring::SelectionCriteria &criteria, Event-Monitoring::EventMonitor_ptr monitor)

  Deletes a Monitoring Task from list of subscriptions.

• **void** `adapt_sampling_rate` (const EventMonitoring::SamplingAddress &address, const EventMonitoring::SelectionCriteria &criteria, Event-Monitoring::Direction direction)

  Applies an offset to the delay value of the appropriate sampling thread.

• **void** `get_monitors` (EventMonitoring::MonitorList_out nodes)

  Gets event channel information about this EventSampler(p. 106).

• **void** `shutdown`()

  Invokes stop.

**Static Private Member Functions**

• **bool** `publishLoop` (void *param)

  A loop that periodically updates information in IS.

**Private Attributes**

• **OWLMutex** `mutex`

  A mutual exclusion object for internal use.

• **IPCAlarm** `publishing_`

  An IPCAlarm, which is used for publishing IS information.

• **PushSamplingFactory** `push_factory_`

  A factory object for the push sampling.

• **PullSamplingFactory** `pull_factory_`

  A factory object for the pull sampling.
• **SamplingAddress address**
  The sampling address of this `EventSampler`(p. 106).

• `unsigned long max_criteria`
  The maximum number of criteria this `EventSampler`(p. 106) will sample.

• **SamplingChannels channels**
  A STL map containing all sampling addresses and selection criteria along with the appropriate `EventChannel`(p. 78) object reference.

• **CannotInitialize * exception**
  An exception that might have occurred in the constructor.

• **EventMonitoring::EventSampler _ var self**
  A CORBA object reference to this `EventSampler`(p. 106).

### A.15.1 Detailed Description

Main class for the `EventSampler`(p. 106) application.

This is the main class for the `EventSampler`(p. 106) application, implementing functionality that has been defined in the IDL declaration of Event-Monitoring::EventSampler. It uses multithreaded behavior so the code needs to be thread aware. The main objective of this class in creation, deletion and management of EventChannels.

**Author:**
Ingo Scholtes

### A.15.2 Constructor & Destructor Documentation

#### A.15.2.1 `emon::EventSampler_impl::EventSampler_impl (const IPCPartition & partition, const SamplingAddress & address, PullSamplingFactory * factory, unsigned long max_criteria = 100)`

Constructor for pull model.

Calls `constructor()`(p. 111) for initialization

**Parameters:**
- `partition` the IPCPartition to start this `EventSampler`(p. 106) in
- `address` the sampling address of this `EventSampler`(p. 106)
- `factory` a reference to the user's pull sampling factory object which will be used to pull events from `EventChannel`(p. 78)
\textit{max criteria} the maximum amount of direct Monitoring Task connections this \texttt{EventSampler} (p. 106) will accept.

See also:
- \texttt{PullSampling} (p. 124)
- \texttt{PullSamplingFactory} (p. 126)
- \texttt{constructor} (p. 111)

\textbf{A.15.2.2} \texttt{emon::EventSampler_impl::EventSampler_impl} (const IPCPartition & \texttt{partition}, const SamplingAddress & \texttt{address}, PushSamplingFactory * \texttt{factory}, unsigned long \texttt{max_criteria} = 100)

Constructor for push model.

Calls method \texttt{constructor} (p. 111) for initialization.

**Parameters:**
- \texttt{partition} the IPCPartition to start this \texttt{EventSampler} (p. 106) in
- \texttt{address} the sampling address of this \texttt{EventSampler} (p. 106)
- \texttt{factory} a reference to the users push sampling factory which will be used to create push sampling threads upon request
- \texttt{max_criteria} the maximum amount of direct Monitoring Task connections this \texttt{EventSampler} (p. 106) will accept

See also:
- \texttt{constructor PushSampling} (p. 128) \texttt{PushSamplingFactory} (p. 130) \texttt{constructor} (p. 111)

\textbf{A.15.3} Member Function Documentation

\textbf{A.15.3.1} \texttt{void emon::EventSampler_impl::adapt_sampling_rate} (const EventMonitoring::SamplingAddress & \texttt{address}, const EventMonitoring::SelectionCriteria & \texttt{criteria}, EventMonitoring::Direction \texttt{direction}) [private]

Applies an offset to the delay value of the appropriate sampling thread.

This method is invoked by the \texttt{Conductor}, in order to adjust the speed at which we sample and ship events to the speed at which the root Monitoring Task connected to this \texttt{EventSampler} (p. 106) can handle them. Depending on the direction, it will increase or decrease the delay in the sampling thread. If the event channel cannot speed up anymore, it will throw \texttt{MaximumReached} to notify the conductor, that he should not bother us anymore with speedup \texttt{Adapt} calls. In any other case we return true.

**Parameters:**
- \texttt{address} the sampling address, which defines origin of the sampled events
criteria the selection criteria of the sampling thread, whose speed we want to adjust

direction whether to speed up or slow down the sampling rate

See also:
EventMonitoring::MaximumReached
EventMonitoring::NotFound
EventMonitoring::Direction

A.15.3.2 void emon::EventSampler_impl::connect_monitor
(const EventMonitoring::SamplingAddress & address,
const EventMonitoring::SelectionCriteria & criteria,
EventMonitoring::EventMonitor_ptr monitor) [private]

Connects a Monitoring Task to this EventSampler(p.106).
This method adds another Monitoring Task to the subscribers list and updates information in IS server. Besides that it will start a new sampling thread, depending on the model of the associated sampling object. If pull model is active, then this method will start the internal pullSamplingThread of class EventChannel(p.78) on the PullSampling(p.124) object created by the factory. If push model is active, it will create a PushSampling(p.128) object. It is up to the users responsibility to ensure, that the creation of this object itself will start a sampling thread. This is an IPC published method.

Parameters:
address the sampling address, which defines origin of the sampled events
criteria the selection criteria the sampled events should satisfy
monitor a CORBA reference to the Monitoring Task to ship events to
(the root Monitoring Task)

Exceptions:
AlreadyConnected
BadAddress
BadCriteria
NoResources

A.15.3.3 void emon::EventSampler_impl::constructor () [private]

This method initializes the sampling process, by connecting the EventSampler(p.106) to the Conductor and publishing the EventSampler(p.106)
A.15.3.4 void emon::EventSampler_impl::destroy ()

Stops all sampling threads, unregisters the EventSampler (p.106) and destroys the IPCObject.

This will stop all sampling threads, unregister the EventSampler (p.106) in the Conductor (if possible) and destroy the IPCObject. It is guaranteed that all sampling threads have been terminated as soon as this method returns.

A.15.3.5 void emon::EventSampler_impl::disconnect_monitor
(const EventMonitoring::SamplingAddress & address,
const EventMonitoring::SelectionCriteria & criteria,
EventMonitoring::EventMonitor_ptr monitor) [private]

Deletes a Monitoring Task from list of subscriptions.

This method deletes one of the root Monitoring Tasks from the subscription list. It will then stop the associated sampling process. This is an IPC published method.

Parameters:
address the sampling address, which defines origin of the sampled events
criteria the selection criteria the sampled events should satisfy
monitor a CORBA reference to the Monitoring Task to ship events to
(the root Monitoring Task)

See also:
NotConnected

A.15.3.6 void emon::EventSampler_impl::get_monitors
(EventMonitoring::MonitorList_out nodes) [private]

Gets event channel information about this EventSampler (p.106).

Returns event channel information about this EventSampler (p.106), requested by a Conductor that has crashed and restarted. When starting up the Conductor, it will find all EventSamplers published in the partition and will ask them about selection criteria, root Monitoring Tasks and sampling address.

Parameters:
nodes a sequence of MonitorList structs, containing information about all Monitoring Tasks currently connected to this EventSampler (p.106)

See also:
EventMonitoring::MonitorList
A.15.3.7  void emon::EventSampler_impl::ping ()  [private]

Used as a proof that this EventSampler(p.106) is alive.
This method is invoked by the Conductor in reasonable interval, in order to
detect possible crashes of EventSamplers. This is an IPC published method.

A.15.3.8  void emon::EventSampler_impl::replace_monitor
(const EventMonitoring::SamplingAddress & address,
 const EventMonitoring::SelectionCriteria & criteria,
 EventMonitoring::EventMonitor_ptr monitor)  [private]

Replaces an existing root Monitoring Task with the new one.
This methods replaces one of the existing Monitoring Tasks in the subscribers
list and updates information in IS server. This is an IPC published method.

Parameters:
- address the sampling address, which defines origin of the sampled events
- criteria the selection criteria the sampled events should satisfy
- monitor a CORBA reference to the Monitoring Task to ship events to
  (the root Monitoring Task)

Exceptions:
- NotConnected

A.15.4  Member Data Documentation

A.15.4.1  unsigned long emon::EventSampler_impl::max_criteria_
[private]

The maximum number of criteria this EventSampler(p.106) will sample.
This is also the maximum number of event channels and sampling threads that
may be active in this EventSampler(p.106) application

The documentation for this class was generated from the following files:

- EventSampler_impl.h
- EventSampler_impl.cc
A.16  emon::MaskedValue Struct Reference

A class representing a masked value.
#include <Common.h>

Public Member Functions

- **MaskedValue** (long value, bool ignore=false)
  
  *Constructor.*

- **MaskedValue** (const EventMonitoring::MaskedValue &mv)
  
  *Copy constructor.*

A.16.1 Detailed Description

A class representing a masked value.

This class represents a masked value. It consists of a long value specifying the value of the selection criteria value and a boolean flag telling whether this value shall be ignored. This class inherits the class generated from the IDL declaration of EventMonitoring::MaskedValue

**Author:**

- Sergei Kolos
- Ingo Scholtes

A.16.2 Constructor & Destructor Documentation

A.16.2.1 emon::MaskedValue::MaskedValue (long value, bool ignore = false)

*Constructor.*

Creates a new **MaskedValue**(p. 114)

**Parameters:**

- **value** the long value to use for this masked value
- **ignore** whether or not to ignore this value

A.16.2.2 emon::MaskedValue::MaskedValue (const EventMonitoring::MaskedValue & mv)

*Copy constructor.*

Creates a copy of masked value mv
Parameters:
  \( mv \) the masked value to copy

The documentation for this struct was generated from the following files:

- Common.h
- Common.cc
A.17 MonitorInfoNamed Class Reference

Contains information about an instance of an active Monitoring Task.
#include <MonitorInfoNamed.h>

Public Member Functions

- MonitorInfoNamed (const IPCPartition &partition, const std::string &name)

Public Attributes

- unsigned long eventsSampled
- unsigned long childMonitors
- bool active
- unsigned long bufferLimit
- unsigned long bufferOccupancy

Protected Member Functions

- MonitorInfoNamed (const IPCPartition &partition, const std::string &name, const std::string &type)
- void publishGuts (ISstream &out)
- void refreshGuts (ISstream &in)

Private Member Functions

- void initialize ()

A.17.1 Detailed Description

Contains information about an instance of an active Monitoring Task.

Author:
IS code generation tool

A.17.2 Member Data Documentation

A.17.2.1 bool MonitorInfoNamed::active

Whether this Monitoring Task is currently sampling events or not
A.17.2.2 unsigned long MonitorInfoNamed::bufferLimit
Maximum number of events to be buffered

A.17.2.3 unsigned long MonitorInfoNamed::bufferOccupancy
Number of events currently buffered

A.17.2.4 unsigned long MonitorInfoNamed::childMonitors
Number of active subscriptions from child Monitoring Tasks

A.17.2.5 unsigned long MonitorInfoNamed::eventsSampled
Total number of sampled events
The documentation for this class was generated from the following file:

- MonitorInfoNamed.h
A.18  emon::MonitorNode Class Reference

Represents a monitor node in the Conductor's local representation of the Monitoring Tree.

#include <MonitorNode.h>

Public Member Functions

- MonitorNode (EventMonitoring::EventMonitor_ptr monitor, unsigned int max_children=1, bool restore=false)
  Constructor.

- EventMonitoring::EventMonitor_ptr monitor () const
  Returns the CORBA object reference to this Monitoring Task.

- const std::vector<MonitorNode *> & children () const
  Returns a STL vector of all children Monitoring Tasks.

- void addChild (EventMonitoring::EventMonitor_ptr monitor, bool restore=false)
  Adds a new child to the Monitoring Task represented by this MonitorNode(p. 118).

- void addChild (MonitorNode *monitor)
  Adds a new child to the Monitoring Task represented by this MonitorNode(p. 118).

- MonitorNode * removeChild (EventMonitoring::EventMonitor_ptr child)
  Removes a child from the children list of this Monitoring Task.

- unsigned int subTreeSize () const
  Returns the size of the subtree of this Monitoring Task.

- MonitorNode * newRootNode ()
  Chooses a new root Monitor Node among the child Monitoring Tasks of this Monitoring Task.

Private Attributes

- EventMonitoring::EventMonitor_var monitor_
  A CORBA object reference to the Monitoring Task.

- unsigned int max_children_
  The maximum amount of children, this Monitoring Task may have.
• unsigned int next_child_
  
  The index of the child that will next receive an addChild invocation if this 
  Monitoring Task can’t handle another child itself.

• std::vector< MonitorNode * > children_
  
  A vector containing pointers to MonitorNodes representing the children of 
  this Monitoring Task.

A.18.1 Detailed Description

Represents a monitor node in the Conductor’s local representation of the Monitoring Tree.

An instance of type MonitorNode (p. 118) contains all necessary information about a Monitoring Task that is connected somewhere in the Monitoring Tree.

It contains a list of it’s children, as well as valid CORBA object reference that might be used for direct communication with the node.

Author:

  Ingo Scholtes

A.18.2 Constructor & Destructor Documentation

A.18.2.1 void eMon::MonitorNode::addChild (MonitorNode * monitor, unsigned int 

  max_children = 1, bool restore = false)

  Constructor.

  Creates a new MonitorNode (p. 118) representing a Monitoring Task.

Parameters:

  monitor a CORBA object reference to the associated Monitoring Task

  max_children the maximum number of children this Monitoring Task may have

  restore whether or not to use restore mode, where the Monitoring Tree is restored using DFS

A.18.3 Member Function Documentation

A.18.3.1 void eMon::MonitorNode::addChild (MonitorNode * monitor)

  Adds a new child to the Monitoring Task represented by this MonitorNode (p. 118).
Parameters:

\[\text{monitor}\] the \text{MonitorNode}(p.118) to add as child to this Monitoring Task

A.18.3.2 \textbf{void emon::MonitorNode::addChild (EventMonitoring::EventMonitor_ptr monitor, bool restore = false)}

Adds a new child to the Monitoring Task represented by this \text{MonitorNode}(p.118).

Parameters:

\[\text{monitor}\] a CORBA object reference of the child to add

\[\text{restore}\] whether or not to use restore mode, i.e. whether or not to notify the Monitoring Tasks about new children

A.18.3.3 \textbf{const std::vector<MonitorNode*> & emon::MonitorNode::children () const [inline]}

Returns a STL vector of all children Monitoring Tasks.

Returns:

a vector of all children Monitoring Task connected to this Monitor Task

A.18.3.4 \textbf{EventMonitoring::EventMonitor_ptr emon::MonitorNode::monitor () const [inline]}

Returns the CORBA object reference to this Monitoring Task.

Returns:

a CORBA object reference to this Monitoring Task

A.18.3.5 \textbf{MonitorNode * emon::MonitorNode::newRootNode ()}

Chooses a new root Monitor Node among the child Monitoring Tasks of this Monitoring Task.

Returns:

a \text{MonitorNode}(p.118) object reference representing the new root Monitoring Task
A.18.3.6 unsigned int emon::MonitorNode::subTreeSize () const

Returns the size of the subtree of this Monitoring Task.

Returns:
the number of Monitoring Tasks in the subtree of this Monitoring Task

The documentation for this class was generated from the following files:

- MonitorNode.h
- MonitorNode.cc
A.19  emon::NoMoreEvents Struct Reference

Exception being thrown by nextEvent or tryNextEvent.
#include <Common.h>

A.19.1  Detailed Description

Exception being thrown by nextEvent or tryNextEvent.
This exception will be thrown by nextEvent or tryNextEvent in case the event buffer is empty. It will be thrown by tryNextEvent every time the buffer is empty and it will be thrown by nextEvent in case the timeout was exceeded and there is still no event in the buffer the EventIterator(p. 92) may return.

See also:
   EventIterator::nextEvent(p. 93)  EventIterator::tryNextEvent(p. 94)

Author:
   Ingo Scholtes

The documentation for this struct was generated from the following file:

   • Common.h
A.20  emon::NoResources Struct Reference

Exception being thrown by emon::select.

#include <Common.h>

Inheritance diagram for emon::NoResources:

```
  emon::CannotInitialize
    |
    v
  emon::NoResources
```

Public Member Functions

- void `raise()`

Public Attributes

- const std::string `what`

A.20.1 Detailed Description

Exception being thrown by emon::select.

This exception will be thrown by emon::select, in case the `EventSampler` (p. 106) does not accept any more connections. This usually means that the maximum number of event channels would be exceeded.

See also:
- emon::select

Author:
- Ingo Scholtes

The documentation for this struct was generated from the following file:

- Common.h
A.21 emon::PullSampling Struct Reference

Abstract class to be used by the user to implement pull model EventSamplers.

#include <PullSampling.h>

Inheritance diagram for emon::PullSampling:

```
+---------------------------------+
| emon::PullSampling              |
+---------------------------------+
|                                 |
+---------------------------------+
| MyPullSampling                  |
```

Public Member Functions

- virtual ~PullSampling ()
  Destructor.

- virtual void sampleEvent (emon::EventChannel &ec)=0
  Sample one event from the hardware and push it to the Monitoring Tasks for distribution.

A.21.1 Detailed Description

Abstract class to be used by the user to implement pull model EventSamplers.

This class, containing a pure virtual function can be used by the user to implement custom EventSamplers following the pull model. Objects of the Pull-Sampling (p.124) inheriting type will be automatically created by the Pull-SamplingFactory (p.126) whenever a subscription of a Monitoring Task arrives and a new EventChannel(p.78) is created.

See also:
  PullSamplingFactory(p.126)

Author:
  Ingo Scholtes

A.21.2 Constructor & Destructor Documentation

A.21.2.1 virtual emon::PullSampling::~PullSampling () [inline, virtual]

Destructor.

The object is destroyed when the sampling activity has to be stopped because it is no more necessary.
A.21.3 Member Function Documentation

A.21.3.1 virtual void emon::PullSampling::sampleEvent
(emon::EventChannel & ec) [pure virtual]

Sample one event from the hardware and push it to the Monitoring Tasks for
distribution.

This virtual method has to be overwritten in order to be able to use the Event-
Monitoring framework.

Parameters:
  ec EventChannel(p. 78) object, which shall be used to push events to the
  connected Monitoring Tasks

See also:
  EventChannel(p. 78)

Implemented in MyPullSampling (p. 141).
The documentation for this struct was generated from the following file:

  * PullSampling.h
A.22 emon::PullSamplingFactory Struct Reference

Abstract base class for user's custom pull factory.

#include <PullSamplingFactory.h>

Inheritance diagram for emon::PullSamplingFactory::

```
+ emon::PullSamplingFactory
|   |
|   v
| MyPullSamplingFactory
```

Public Member Functions

- virtual ~PullSamplingFactory ()

  Virtual destructor.

- virtual PullSampling * startSampling (const SamplingAddress &sa,
  const SelectionCriteria &sc)=0 throw (BadAddress, BadCriteria, No-
  Resources)

  Virtual method, implementation shall create a new PullSampling(p. 124)
  object upon request.

A.22.1 Detailed Description

Abstract base class for user's custom pull factory.

This abstract base class shall be used by the user in order to implement a custom
factory to create objects of type PullSampling(p. 124)

Author:
  Ingo Scholtes

See also:
  PullSampling(p. 124)
A.22.2 Member Function Documentation

A.22.2.1 virtual PullSampling::emon::PullSamplingFactory::start-
Sampling (const SamplingAddress & sa, const
SelectionCriteria & sc) throw (BadAddress, BadCriteria,
NoResources) [pure virtual]

Virtual method, implementation shall create a new PullSampling(p. 124) ob-
ject upon request.

This method is invoked, whenever a new sampling thread is started. Each new
thread will be associated with a new PullSampling(p. 124) object.

Parameters:

- sa SamplingAddress(p. 135) for this sampling thread
- sc selection criteria the events sampled by this thread will satisfy

Returns:

- a reference to a new PullSampling(p. 124) object responsible for event
sampling

Exceptions:

- emon::BadAddress
- emon::BadCriteria
- emon::NoResources

The documentation for this struct was generated from the following file:

- PullSamplingFactory.h
A.23 emon::PushSampling Struct Reference

Abstract class to be used by the user to implement push model Event Samplers.

#include <PushSampling.h>

Inheritance diagram for emon::PushSampling:

```
emon::PushSampling
    ^
   |  
MyPushSampler
```

Public Member Functions

- virtual ~PushSampling ()

  Destructor.

A.23.1 Detailed Description

Abstract class to be used by the user to implement push model Event Samplers. An object of this class has to be created by the PushSamplingFactory (p. 130) when sampling is requested to be started. This object will be automatically destroyed when sampling has to be stopped. When using this class, the user will have to ensure, that a new thread is started upon object creation and cleanly exited upon object destruction. It is the user’s responsibility to push events to the EventChannel (p. 78) using his custom thread.

See also:
  PushSamplingFactory (p. 130)

Author:
  Ingo Scholtes

A.23.2 Constructor & Destructor Documentation

A.23.2.1 virtual emon::PushSampling::~PushSampling () [inline, virtual]

  Destructor.

  The object is destroyed when the sampling activity has to be stopped because it is no more necessary.

  The documentation for this struct was generated from the following file:
- PushSampling.h
A.24  emon::PushSamplingFactory Struct Reference

Abstract base class for user’s custom push factory.

#include <PushSamplingFactory.h>

Inheritance diagram for emon::PushSamplingFactory:

```
emon::PushSamplingFactory

MyPushSamplingFactory
```

Public Member Functions

- virtual ~PushSamplingFactory ()
  
  *Virtual destructor.*

- virtual PushSampling * startSampling (const SamplingAddress &sa, const SelectionCriteria &sc, EventChannel *channel)=0 throw (BadAddress, BadCriteria, NoResources)

  *Virtual method, creates a new PushSampling(p. 128) object upon request.*

A.24.1 Detailed Description

Abstract base class for user’s custom push factory.

This class shall be used by the user in order to implement a custom factory to create objects of type PushSampling(p. 128)

Author:  
  Ingo Scholtes

See also:  
  PushSampling(p. 128)

A.24.2 Member Function Documentation

A.24.2.1 virtual PushSampling* emon::PushSamplingFactory::startSampling (const SamplingAddress & sa, const SelectionCriteria & sc, EventChannel * channel) throw (BadAddress, BadCriteria, NoResources) [pure virtual]

Virtual method, creates a new PushSampling(p. 128) object upon request.
This method is invoked, whenever a new sampling thread is started. Each new thread will be associated with a new \texttt{PushSampling}(p. 128) object.

\textbf{Parameters:}
\begin{itemize}
  \item \texttt{sa} \texttt{SamplingAddress}(p. 135) for this sampling thread
  \item \texttt{sc} \texttt{SelectionCriteria}(p. 137) the events sampled by this thread will satisfy
  \item \texttt{channel} an \texttt{EventChannel}(p. 78) object that may be used to push events to Monitoring Tasks
\end{itemize}

\textbf{Returns:}
\begin{itemize}
  \item a reference to a new \texttt{PushSampling}(p. 128) object responsible for event sampling
\end{itemize}

\textbf{Exceptions:}
\begin{itemize}
  \item \texttt{emon::BadAddress}
  \item \texttt{emon::BadCriteria}
  \item \texttt{emon::NoResources}
\end{itemize}

The documentation for this struct was generated from the following file:
\begin{itemize}
  \item \texttt{PushSamplingFactory.h}
\end{itemize}
A.25 SamplerInfoNamed Class Reference

Contains information about an instance of an active EventSampler.

```
#include <SamplerInfoNamed.h>
```

**Public Member Functions**

- `SamplerInfoNamed` (const IPCPartition &partition, const std::string &name)

**Public Attributes**

- unsigned long `eventsSampled`
- unsigned long `rootMonitorSwaps`
- unsigned long `activeChannels`
- unsigned long `maxConnections`
- unsigned long `totalSubscriptions`
- unsigned long `speedUpRequests`
- unsigned long `slowDownRequests`

**Protected Member Functions**

- `SamplerInfoNamed` (const IPCPartition &partition, const std::string &name, const std::string &type)
- void `publishGuts` (Iostream &out)
- void `refreshGuts` (IStream &in)

**Private Member Functions**

- void `initialize` ()

A.25.1 Detailed Description

Contains information about an instance of an active EventSampler.

**Author:**

IS code generation tool

A.25.2 Member Data Documentation

A.25.2.1 unsigned long `SamplerInfoNamed::activeChannels`

Number of active subscriptions from Monitoring Tasks
A.25.2.2 unsigned long SamplerInfoNamed::eventsSampled
Total number of sampled events

A.25.2.3 unsigned long SamplerInfoNamed::maxConnections
Maximum number of Monitoring Task connections accepted

A.25.2.4 unsigned long SamplerInfoNamed::rootMonitorSwaps
total number of root Monitoring Task swaps

A.25.2.5 unsigned long SamplerInfoNamed::slowDownRequests
Total number of slowdown requests received from monitors

A.25.2.6 unsigned long SamplerInfoNamed::speedUpRequests
Total number of speedup requests received from Monitoring Tasks

A.25.2.7 unsigned long SamplerInfoNamed::totalSubscriptions
Total number of subscriptions received from Monitoring Tasks
The documentation for this class was generated from the following file:

- SamplerInfoNamed.h
A.26 emon::SamplerStopped Struct Reference

Exception being thrown by nextEvent or tryNextEvent.
#include <Common.h>

A.26.1 Detailed Description

Exception being thrown by nextEvent or tryNextEvent.

This exception will be thrown by nextEvent or tryNextEvent, in case the EventSampler(p.106) exited or crashed.

See also:

EventIterator::nextEvent(p.93) EventIterator::tryNextEvent(p.94)

Author:

Ingo Scholtes

The documentation for this struct was generated from the following file:

- Common.h
A.27  emon::SamplingAddress Struct Reference

A class representing a sampling address.
#include <Common.h>

Public Member Functions

- SamplingAddress ()
  Constructor.

- SamplingAddress (const EventMonitoring::SamplingAddress &sa)
  Copy constructor.

- void addComponent (const std::string &key, const std::string &value)
  Adds an address component to this sampling address.

A.27.1 Detailed Description

A class representing a sampling address.

This class represents a sampling address consisting of a sequence of address components. It inherits the class generated from the IDL declaration of EventMonitoring::SamplingAddress

Author:
  Sergei Kolos
  Ingo Scholtes

A.27.2 Constructor & Destructor Documentation

A.27.2.1 emon::SamplingAddress::SamplingAddress ()

Constructor.
Default empty constructor

A.27.2.2 emon::SamplingAddress::SamplingAddress (const EventMonitoring::SamplingAddress & sa)

Copy constructor.
Creates a copy of sampling address sa

Parameters:
  sa the sampling address to copy
A.27.3 Member Function Documentation

A.27.3.1 void emon::SamplingAddress::addComponent (const std::string & key, const std::string & value)

Adds an address component to this sampling address.

This method adds an address component consisting of key and value to this sampling address

Parameters:

- **key** the key of the address component to add
- **value** the value of the key of the address component to add

The documentation for this struct was generated from the following files:

- Common.h
- Common.cc
A.28  emon::SelectionCriteria Struct Reference

A class representing a selection criteria.

#include <Common.h>

Public Member Functions

- SelectionCriteria (MaskedValue detector_type, MaskedValue lvl1_trigger_type, MaskedValue lvl2_trigger_info, MaskedValue status_word, long statistics=0)
  Constructor.

- SelectionCriteria (long detector_type, long lvl1_trigger_type, long lvl2_trigger_info, long status_word, long statistics=0)
  Constructor.

- SelectionCriteria (const EventMonitoring::SelectionCriteria &sc)
  Copy constructor.

A.28.1 Detailed Description

A class representing a selection criteria.

A.28.2 Constructor & Destructor Documentation

A.28.2.1  emon::SelectionCriteria::SelectionCriteria (MaskedValue
detector_type, MaskedValue lvl1_trigger_type,
MaskedValue lvl2_trigger_info, MaskedValue
status_word, long statistics = 0)

Constructor.

This may be used if you want to use masked values

Parameters:

detector_type MaskedValue(p. 114) specifying the type of the detector
lvl1_trigger_type MaskedValue(p. 114) specifying the type of the
level 1 trigger
lvl2_trigger_info MaskedValue(p. 114) specifying information for
level 2 trigger
status_word MaskedValue(p. 114) specifying status word in event
header

statistics optional long parameter, specifying how many events to skip
between two events that shall be sampled. a value of x means: sample
every xth event
A.28.2.2  `emon::SelectionCriteria::SelectionCriteria` (long
        `detector_type`, long lvl1_trigger_type, long
        lvl2_trigger_info, long status_word, long statistics = 0)

Constructor.

This may be used if you do not want to mask any values

**Parameters:**

- **detector_type** long value specifying the type of the detector
- **lvl1_trigger_type** long value specifying the type of the level1 trigger
- **lvl2_trigger_info** long value specifying information for level2 trigger
- **status_word** long value specifying status word in event header
- **statistics** optional parameter, specifying how many events to skip between
two events that shall be sampled. a value of x means: sample every
xth event

A.28.2.3  `emon::SelectionCriteria::SelectionCriteria` (const
        EventMonitoring::SelectionCriteria & sc)

Copy constructor.

Creates a copy of selection criteria sc

**Parameters:**

- **sc** the selection criteria to copy

The documentation for this struct was generated from the following files:

- Common.h
- Common.cc
Appendix B

ATLAS Event Monitoring System Examples

B.1 MyPullSampling Class Reference

Pull model EventSampler implementation class.
Inheritance diagram for MyPullSampling:

```
emon::PullSampling
  MyPullSampling
```

Public Member Functions

- **MyPullSampling** (const emon::SelectionCriteria &sc, unsigned long *event, long size)
  Constructor of the example implementation.

- void **sampleEvent** (emon::EventChannel &cc)
  Example implementation of the sampling process.

- ~**MyPullSampling** ()
  Destructor of the example implementation.

Private Attributes

- unsigned long * **event**
example event data

• long size_
  size of the example events

B.1.1 Detailed Description

Pull model EventSampler implementation class.

This is an example of how to use the abstract PullSampling class. Apart from the constructor, you will have to override the method sampleEvent. The startSampling method of the PullSamplingFactory shall return instances of this class. For every `emon::EventChannel` (p. 78) one instance of this class will be created.

Author:
  Ingo Scholtes
  Sergei Kolos

See also:
  emon::PullSampling (p. 124)  emon::PullSamplingFactory (p. 126)
  emon::EventSampler (p. 106)  emon::EventChannel (p. 78)

B.1.2 Constructor & Destructor Documentation

B.1.2.1 MyPull Sampling::MyPull Sampling (const emon::SelectionCriteria & sc, unsigned long * event, long size) [inline]

Constructor of the example implementation.

In real life, this method usually initializes hardware of the data flow system for event sampling.

Parameters:
  sc the selection criteria sampled events shall match
  event example event to push to the event channel
  size size of the example event

B.1.2.2 MyPull Sampling::~MyPull Sampling () [inline]

Destructor of the example implementation.

In real life, this method usually deallocates memory and prepares data flow for shutdown. Whenever an event channel will be closed, the appropriate destructor will be called.
B.1.3 Member Function Documentation

B.1.3.1 void MyPullSampling::sampleEvent (emon::EventChannel & cc) [inline, virtual]

Example implementation of the sampling process.
This method just pushes the example event to the Monitoring Tasks. When
pushEvent of EventChannel has finished, it is guaranteed, that the event buffer
passed to EventChannel may be freed.

See also:
  EventChannel

Implements emon::PullSampling (p. 125).
The documentation for this class was generated from the following file:

  • PullSamplerMain.cc
B.2 MyPullSamplingFactory Class Reference

Example implementation of the PullSamplingFactory.

Inheritance diagram for MyPullSamplingFactory::

```
emon::PullSamplingFactory

MyPullSamplingFactory
```

Public Member Functions

- **MyPullSamplingFactory** (unsigned long *event, long size)
  
  Constructor.

- **emon::PullSampling * startSampling** (const emon::SamplingAddress &address, const emon::SelectionCriteria &criteria) throw (emon::BadAddress, emon::BadCriteria, emon::NoResources)
  
  *This will be called whenever a new EventChannel needs to be created.*

- **virtual PullSampling * startSampling** (const SamplingAddress &sa, const SelectionCriteria &sc)=0 throw (BadAddress, BadCriteria, NoResources)
  
  *Virtual method, implementation shall create a new PullSampling(p. 124) object upon request.*

Private Attributes

- unsigned long *event_
  
  *example event data*

- long size_
  
  *size of the example events*

B.2.1 Detailed Description

Example implementation of the PullSamplingFactory.

This class is an example implementation of the PullSamplingFactory, its main purpose being the creation of MyPullSampling(p. 139) objects in the method startSampling.

**Author:**

Ingo Scholtes
B.2.2 Constructor & Destructor Documentation

B.2.2.1 MyPullSamplingFactory::MyPullSamplingFactory
   (unsigned long * event, long size) [inline]

Constructor.
This example constructor simply allocates some memory for a new example event or uses the example event passed in the constructor.

Parameters:
   event the example event that shall be pushed to the EventChannel
   size size of the example event

B.2.3 Member Function Documentation

B.2.3.1 virtual PullSampling* emon::PullSamplingFactory::start-Sampling (const SamplingAddress & sa, const
   SelectionCriteria & sc) throw (BadAddress, BadCriteria, NoResources) [pure virtual, inherited]

Virtual method, implementation shall create a new PullSampling(p. 124) object upon request.
This method is invoked, whenever a new sampling thread is started. Each new thread will be associated with a new PullSampling(p. 124) object.

Parameters:
   sa SamplingAddress(p. 135) for this sampling thread
   sc selection criteria the events sampled by this thread will satisfy

Returns:
   a reference to a new PullSampling(p. 124) object responsible for event sampling

Exceptions:
   emon::BadAddress
   emon::BadCriteria
   emon::NoResources

B.2.3.2 emon::PullSampling* MyPullSamplingFactory::start-Sampling (const emon::SamplingAddress &
   address, const emon::SelectionCriteria & criteria)
   throw (emon::BadAddress, emon::BadCriteria, emon::NoResources) [inline]

This will be called whenever a new EventChannel needs to be created.
This method will be called whenever a new EventChannel is required and a new sampling thread needs to be created. It’s main purpose is to enable users to pass custom parameters to the **MyPullSampling** (p. 139) objects upon object creation time.

**Parameters:**
- **address** the sampling address of this application
- **criteria** the selection criteria, the events being sampled shall satisfy

**Returns:**
a new instance of class **MyPullSampling** (p. 139)

**See also:**
- **MyPullSampling** (p. 139)  
- **emon::PullSampling** (p. 124)  
- **emon::EventChannel** (p. 78)

The documentation for this class was generated from the following file:

- PullSamplerMain.cc
B.3 MyPushSampler Class Reference

Example PushSampling implementation.
Inheritance diagram for MyPushSampler::

```
emon::PushSampling
   |
   v
MyPushSampler
```

Public Member Functions

- **MyPushSampler** (const emon::SelectionCriteria &sc,
  emon::EventChannel *channel, unsigned long *event, long size)
  
  Constructor of the example implementation.

- **void * run_undetached** (void *)
  
  This method will be executed in a separate thread.

- **~MyPushSampler** ()
  
  Destructor of the example implementation.

Private Attributes

- **bool terminated**
  
  Tells whether our thread should be terminated.

- **emon::EventChannel * channel**
  
  A reference to the Event Channel.

- **OWLMutex term_mutex**
  
  A mutual exclusion for the termination condition variable.

- **OWLCondition term_condition**
  
  A condition variable ensuring that the thread will be terminated before the
  object is destroyed.

- **unsigned long * event**
  
  Example event data.

- **long size**
  
  Size of the example events.
B.3.1 Detailed Description

Example PushSampling implementation.

This is an example of how to use the abstract PushSampling class. You only will have to override the constructor and destructor of this class. In the constructor you will have to create a new thread pushing events to the EventChannels passed by the constructor. In the destructor you will have to make sure that the thread you creates cleanly exits.

Author:
Ingo Scholtes

See also:
emon::PushSampling(p.128)  emon::EventSampler(p.106)
emon::PushSamplingFactory(p.130)  emon::EventChannel(p.78)

B.3.2 Constructor & Destructor Documentation

B.3.2.1 MyPushSampler::MyPushSampler (const
eleon::SelectionCriteria & sc, emon::EventChannel *
channel, unsigned long * event, long size) [inline]

Constructor of the example implementation.

This example implementation simply starts a thread and uses custom parameters to pass an example event in the constructor.

Parameters:

- sc the selection criteria the sampled events should satisfy
- channel a reference to the emon::EventChannel(p.78) you will want to use in order to push events to Monitoring Tasks
- event an example event
- size the size of the example event

B.3.2.2 MyPushSampler::~MyPushSampler () [inline]

Destructor of the example implementation.

Set the flag for thread termination, wait for the thread to terminate, clean up memory and exit.

B.3.3 Member Function Documentation

B.3.3.1 void* MyPushSampler::run_undetached (void *) [inline]

This method will be executed in a separate thread.
Its only purpose is to push events to the \texttt{emon::EventChannel} (p. 78) object. The documentation for this class was generated from the following file:

- PushSamplerMain.cc
B.4 MyPushSamplingFactory Class Reference

Example implementation of the PushSamplingFactory.
Inheritance diagram for MyPushSamplingFactory:

```
emon::PushSamplingFactory
    
MyPushSamplingFactory
```

Public Member Functions

- `MyPushSamplingFactory` (unsigned long *event, long size)
  *Constructor.*

- `emon::PushSampling * startSampling` (const `emon::SamplingAddress` &address, const `emon::SelectionCriteria` &criteria, `emon::EventChannel` *channel) throw (emon::BadAddress, emon::BadCriteria, emon::NoResources)
  *This will be called whenever a new event channel needs to be created.*

- virtual `PushSampling * startSampling` (const SamplingAddress &sa, const SelectionCriteria &sc, EventChannel *channel)=0 throw (BadAddress, BadCriteria, NoResources)
  *Virtual method, creates a new PushSampling(p. 128) object upon request.*

Private Attributes

- unsigned long *event_
  *Example event data.*

- long size_
  *Size of the example events.*

B.4.1 Detailed Description

Example implementation of the PushSamplingFactory.
This class is an example implementation of the PushSamplingFactory, it's only purpose being the creation of MyPushSampling objects in the method startSampling

**Author:**
Ingo Scholtes
B.4.2 Constructor & Destructor Documentation

B.4.2.1 MyPushSamplingFactory::MyPushSamplingFactory
(unsigned long *event, long size) [inline]

Constructor.
This example constructor simply allocates some memory for a new example event or uses the example event passed in the constructor.

Parameters:
- event the example event to use
- size the size of the example event

B.4.3 Member Function Documentation

B.4.3.1 virtual PushSampling* emon::PushSamplingFactory::start-
Sampling (const SamplingAddress & sa, const
SelectionCriteria & sc, EventChannel * channel) throw
(BadAddress, BadCriteria, NoResources) [pure virtual,
handed]

Virtual method, creates a new PushSampling(p. 128) object upon request.

This method is invoked, whenever a new sampling thread is started. Each new thread will be associated with a new PushSampling(p. 128) object.

Parameters:
- sa SamplingAddress(p. 135) for this sampling thread
- sc SelectionCriteria(p. 137) the events sampled by this thread will satisfy
- channel an EventChannel(p. 78) object that may be used to push events to Monitoring Tasks

Returns:
a reference to a new PushSampling(p. 128) object responsible for event sampling

Exceptions:
- emon::BadAddress
- emon::BadCriteria
- emon::NoResources

B.4.3.2 emon::PushSampling* MyPushSamplingFactory::start-
Sampling (const emon::SamplingAddress & address, const
emon::SelectionCriteria & criteria, emon::EventChannel * channel) throw (emon::BadAddress, emon::BadCriteria,
emon::NoResources) [inline]

This will be called whenever a new event channel needs to be created.
This method will be called whenever a new EventChannel has to be created and it must return a new object of type MyPushSampling, custom parameters may be passed to this object.

**Parameters:**
- **address** the sampling address of this EventSampler
- **criteria** the selection criteria the sampled events should satisfy
- **channel** the emon::EventChannel(p.78) object we need to push events to

**Returns:**
a new instance of MyPushSampling

**See also:**
- emon::PushSampling(p.128) MyPushSampling

The documentation for this class was generated from the following file:

- PushSamplerMain.cc
List of Acronyms

LHC       Large Hadron Collider, 7
PS        Proton Synchroton, 7
SPS       Super Proton Synchroton, 7
DAQ       Data Aquisition, 10
TDAQ      Trigger and Data Aquisition, 10
ROC       Read-Out Crate, 10
DCS       Detector Control System, 10
EB        Event Builder, 10
IPC       Interprocess Communication, 10
IS        Information Service, 11
MRS       Message Reporting System, 11
ROD       Read-Out Drivers, 13
ROS       Read-Out System, 13
CORBA     Common Object Request Broker Architecture, 15
OMG       Object Management Group, 15
ORB       Object request broker, 15
IDL       Interface Definition Language, 15
BOA       Basic Object Adaptor, 15
POA       Portable Object Adaptor, 15
IIOP      Internet Inter ORB Protocol, 16
GIOP      General Inter ORB Protocol, 16
IOR       Interoperable Object Reference, 16
COS       Common Object Services, 17
ROB       Read Out Buffer, 27
OWL       Online Wide Library, 38
DFS       Depth-First Search, 41
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFI</td>
<td>Sub-Farm Input</td>
<td>42</td>
</tr>
<tr>
<td>LST</td>
<td>Large Scale Tests</td>
<td>51</td>
</tr>
<tr>
<td>HLT</td>
<td>High Level Trigger</td>
<td>52</td>
</tr>
<tr>
<td>AFS</td>
<td>Andrew File System</td>
<td>53</td>
</tr>
<tr>
<td>LXPLUS</td>
<td>LimuX Public Login User Service</td>
<td>56</td>
</tr>
<tr>
<td>CVS</td>
<td>Concurrent Versions System</td>
<td>59</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Definitions ........................................ 45
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The LHC with its connected detectors [1]</td>
<td>8</td>
</tr>
<tr>
<td>1.2</td>
<td>The ATLAS Trigger System ([3], p. 6)</td>
<td>9</td>
</tr>
<tr>
<td>1.3</td>
<td>Atlas Online Software Component Decomposition</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>CORBA Architecture</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Distribution sequence diagram (UML notation) [11]</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Getting rid of the bottleneck: n Monitoring Tasks/n Event Samplers</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>Getting rid of the bottleneck: n Monitoring Tasks/1 Event Sampler</td>
<td>22</td>
</tr>
<tr>
<td>4.3</td>
<td>Binary Monitoring Tree Example</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>Distribution sequence diagram (UML notation)</td>
<td>24</td>
</tr>
<tr>
<td>4.5</td>
<td>Collaboration diagram (UML notation)</td>
<td>25</td>
</tr>
<tr>
<td>4.6</td>
<td>Outline of a template class factory API with some pseudocode</td>
<td>26</td>
</tr>
<tr>
<td>4.7</td>
<td>Event Sampler class (Pull API)</td>
<td>28</td>
</tr>
<tr>
<td>4.8</td>
<td>PullSampling and PullSamplingFactory interfaces</td>
<td>28</td>
</tr>
<tr>
<td>4.9</td>
<td>A possible PullSamplingFactory implementation (with some pseudo code)</td>
<td>29</td>
</tr>
<tr>
<td>4.10</td>
<td>Example of the PullSampling interface implementation (with some pseudo code)</td>
<td>29</td>
</tr>
<tr>
<td>4.11</td>
<td>Example of the main function for an Event Sampler application (with some pseudocode)</td>
<td>30</td>
</tr>
<tr>
<td>4.12</td>
<td>Event Sampler class (Push API)</td>
<td>31</td>
</tr>
<tr>
<td>4.13</td>
<td>PushSampling and PushSamplingFactory interfaces</td>
<td>32</td>
</tr>
<tr>
<td>4.14</td>
<td>A possible PushSamplingFactory implementation (with some pseudo code)</td>
<td>32</td>
</tr>
<tr>
<td>4.15</td>
<td>Example of the PushSampling interface implementation (with some pseudo code)</td>
<td>33</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.16 Example of the main function for Event Sampler application (with some pseudocode) .......................... 33
4.17 Monitoring Task entry point ............................................. 34
4.18 Event Iterator interface .................................................. 35
4.19 The Event Monitoring System interfaces (OMG IDL) ............. 36

6.1 Large Scale Test network topology[16] .............................. 52
6.2 Results of Configuration A ............................................. 53
6.3 Results of Configuration B .............................................. 54
6.4 Accumulated throughput over 5 sampling channels for a single Event Sampler ........................................ 55
6.5 Throughput per Monitoring Task ....................................... 57
6.6 Accumulated throughput of multiple concurrent Monitoring Tasks 58
Index

~EventChannel
  emon::EventChannel, 81
~EventMonitor_impl
  emon::EventMonitor_impl, 98
~MyPullSampling
  MyPullSampling, 140
~MyPushSampler
  MyPushSampler, 146
~PullSampling
  emon::PullSampling, 124
~PushSampling
  emon::PushSampling, 128

active
  MonitorInfoNamed, 116
activeChannels
  SamplerInfoNamed, 132
activeMonitors
  ConductorInfoNamed, 74
activeSamplers
  ConductorInfoNamed, 74
adapt_sampling_rate
  emon::Conductor_impl, 68
  emon::EventSampler_impl, 110
adaptAccepted
  ConductorInfoNamed, 74
adaptReceived
  ConductorInfoNamed, 74
adaptSamplingRate
  emon::EventMonitor_impl, 99
add_child
  emon::EventMonitor_impl, 99
add_monitor
  emon::Conductor_impl, 68
addChild
  emon::MonitorNode, 119, 120
addComponent
  emon::SamplingAddress, 136
address
  emon::EventChannel, 82
API, 23

Monitoring Task, 33
Previous Implementation, 19
Sampler (Pull Model), 27
Sampler (Push Model), 31

ATLAS
  detector, 7
  Online Software, 10
  TDAQ System, 10
  Triggers, 8
availableEvents
  emon::EventIterator, 93

bandwidth, 57
BOOST, 38
buffer_
  emon::EventMonitor_impl, 102
bufferLimit
  MonitorInfoNamed, 116
bufferOccupancy
  MonitorInfoNamed, 117

childMonitors
  MonitorInfoNamed, 117
children
  emon::MonitorNode, 120
Client/Server Paradigm, 19, 21
Communication Complexity
  Conductor, 50
  Distributor, 47
  Event Sampler (previous), 45
  Event Sampler (reimplementation), 49
  Monitoring Task (previous), 46
  Monitoring Task(reimplementation), 49

Conductor, 21, 38
Communication Complexity, 50
Crash, 40
Conductor_impl
  emon::Conductor_impl, 67
ConductorInfoNamed, 73
ConductorInfoNamed
  activeMonitors, 74
  activeSamplers, 74
  adaptAccepted, 74
  adaptReceived, 74
  crashedConductors, 74
  crashedSamplers, 74
  previousConductorCrashed, 74
  totalMonitors, 74
  totalSamplers, 74
  treeType, 74
Congestion Control, 41
connect_monitor
  emon::EventSampler_impl, 111
connect_sampler
  emon::Conductor_impl, 69
constructor
  emon::EventSampler_impl, 111
CORBA, 10, 15
  BOA, 15
  COS Naming, 17
  GIOP, 16
  IOR-Strings, 16
  OMG, 15
  ORB, 15
crashedConductors
  ConductorInfoNamed, 74
crashedSamplers
  ConductorInfoNamed, 74
data
  emon::Event, 77
delay
  emon::EventChannel, 82
destroy
  emon::EventMonitor_impl, 99
  emon::EventSampler_impl, 111
DFS, 41
disconnect_monitor
  emon::EventSampler_impl, 112
disconnect_sampler
  emon::Conductor_impl, 69
Distributor
  Communication Complexity, 47
  Communication complexity, 45, 47
emon::BadAddress, 61
emon::BadCriteria, 63
emon::CannotInitialize, 64
emon::Conductor_impl, 65
  adapt_sampling_rate, 68
  add_monitor, 68
  Conductor_impl, 67
  connect_sampler, 69
  disconnect_sampler, 69
  isExist, 69
  max_children_, 71
  monitors_, 71
  pingLoop, 70
  remove_monitor, 70
  removeSampler, 70
  samplers_, 71
  shutdown, 71
  throwInitializationException, 71
emon::Event, 76
  data, 77
  Event, 76
  isContiguous, 77
  size, 77
emon::EventChannel, 78
emon::EventChannel
  ~EventChannel, 81
  address, 82
  delay, 82
  EventChannel, 81
  failed, 82
  monitor, 82
  pingLoop, 82
  pushEvent, 83
  replaceMonitor, 83
  speedUp, 84
  wait, 84
emon::EventChannel::PullSamplingThread, 85
emon::EventChannel::PullSamplingThread
  maybeSuspend, 86
  PullSamplingThread, 86
  run_undetached, 86
  start, 86
  stop, 87
emon::EventChannel::RecoveryThread, 88
emon::EventChannel::RecoveryThread
  RecoveryThread, 89
emon::EventFragment, 90
emon::EventFragment
  EventFragment, 90
eventsDropped
emon::EventIterator, 93

eventsSampled
  MonitorInfoNamed, 117
  SamplerInfoNamed, 132

exit_
  emon::EventMonitor_impl::FwdThread, 105

Factory Pattern, 25, 28, 31
failed
emon::EventChannel, 82

Fault Tolerance, 39
  Conductor, 40
  Heartbeat Protocol, 40
  Monitoring Task, 23, 39
  Previous implementation, 20
  Primary-Backup Approach, 40
  Sampler, 39

get_children
  emon::EventMonitor_impl, 99

get_monitors
  emon::EventSampler_impl, 112

IOR, 41
IPC, 10, 17, 20
IS, 11, 40
isContiguous
  emon::Event, 77
isExist
  emon::Conductor_impl, 69

Large Scale Test, 51
LHC, 7
  Accelerator, 7
  ATLAS Detector, 7
LXPLUS, 56

MaskedValue
  emon::MaskedValue, 114
max_children_
  emon::Conductor_impl, 71
max_criteria_
  emon::EventSampler_impl, 113
maxConnections
  SamplerInfoNamed, 133
maybeSuspend
  emon::EventChannel::PullSamplingThread, 86

Memory

Contiguous, 42
Segmentation, 42

monitor
  emon::EventChannel, 82
  emon::MonitorNode, 120
  MonitorInfoNamed, 116
  MonitorInfoNamed
    active, 116
    bufferLimit, 116
    bufferOccupancy, 117
    childMonitors, 117
    eventsSampled, 117
  Monitoring Task, 12, 19–22, 37
  Communication complexity (previous), 46
  Communication Complexity (reimplementation), 49
  Crash, 23, 39
  Monitoring Tree, 21–23, 45
  MonitorNode
    emon::MonitorNode, 119
    monitors_
      emon::Conductor_impl, 71
  MRS, 11
MyPullSampling, 139
  MyPullSampling, 140
MyPullSampling
  ~MyPullSampling, 140
  MyPullSampling, 140
sampleEvent, 141
MyPullSamplingFactory, 142
  MyPullSamplingFactory, 143
MyPullSamplingFactory
  MyPullSamplingFactory, 143
  startSampling, 143
MyPushSampler, 145
  MyPushSampler, 146
MyPushSampler
  ~MyPushSampler, 146
  MyPushSampler, 146
  run_undetached, 146
  MyPushSamplingFactory, 148
  MyPushSamplingFactory, 149
MyPushSamplingFactory
  MyPushSamplingFactory, 149
  startSampling, 149

newRootNode
  emon::MonitorNode, 120

nextEvent
emon::EventIterator, 93
emon::EventMonitor_impl, 100

OWL, 38

Peer-To-Peer Paradigm, 21
ping
  emon::EventSampler_impl, 112
pingLoop
  emon::Conductor_impl, 70
  emon::EventChannel, 82
POA, 15

previousConductorCrashed
  ConductorInfoNamed, 74
Pull Model, 24, 27

PullSamplingThread
  emon::EventChannel::Pull-SamplingThread, 86
Push Model, 24, 31
push_event
  emon::EventMonitor_impl, 100
pushEvent
  emon::EventChannel, 83

RecoveryThread
  emon::EventChannel::RecoveryThread, 89
remove_child
  emon::EventMonitor_impl, 100
remove_monitor
  emon::Conductor_impl, 70
removeSampler
  emon::Conductor_impl, 70
replace_monitor
  emon::EventSampler_impl, 113
replaceMonitor
  emon::EventChannel, 83
retry_adapt
  emon::EventMonitor_impl, 101
ROD, 27
ROOT, 13
rootMonitorSwaps
  SamplerInfoNamed, 133
ROS, 27
run_undetached
  emon::EventChannel::Pull-SamplingThread, 86
  MyPushSampler, 146
sampleEvent
  emon::PullSampling, 125
  MyPullSampling, 141
sampler_exit
  emon::EventMonitor_impl, 101
SamplerInfoNamed, 132
SamplerInfoNamed
  activeChannels, 132
  eventsSampled, 132
  maxConnections, 133
  rootMonitorSwaps, 133
  slowDownRequests, 133
  speedUpRequests, 133
  totalSubscriptions, 133
samplers_
  emon::Conductor_impl, 71
SamplingAddress
  emon::SamplingAddress, 135
scalability, 56
SelectionCriteria
  emon::SelectionCriteria, 137, 138
shutdown
  emon::Conductor_impl, 71
size
  emon::Event, 77
slowDownRequests
  SamplerInfoNamed, 133
Smart Pointer, 38
Spanning tree, 41
speedUp
  emon::EventChannel, 84
speedUpRequests
  SamplerInfoNamed, 133
start
  emon::EventChannel::Pull-SamplingThread, 86
  emon::EventMonitor_impl, 101
started_
  emon::EventMonitor_impl::Fwd-Thread, 105
startSampling
  emon::PullSamplingFactory, 127
  emon::PushSamplingFactory, 130
MyPullSamplingFactory, 143
MyPushSamplingFactory, 149
stop
  emon::EventChannel::Pull-SamplingThread, 87
  emon::EventMonitor_impl, 101
subTreeSize
  emon::MonitorNode, 120
throwInitializationException
  emon::Conductor_impl, 71
totalMonitors
  ConductorInfoNamed, 74
totalSamplers
  ConductorInfoNamed, 74
totalSubscriptions
  SamplerInfoNamed, 133
treeType
  ConductorInfoNamed, 74
tryNextEvent
  emon::EventIterator, 94

wait
  emon::EventChannel, 84
Bibliography


