ATLAS TDAQ

Error Handling and Error Reporting in TDAQ Applications

Document Version:	1.0
Document ID:	ATL-D-EN-0003
Document Date:	18 December 2004
Document Status:	Draft

Abstract

This document reviews the common needs for a general error handling and reporting system to be used inside ATLAS TDAQ applications. It is envisaged that it shall become the standard tool used in all subsystems that need to report errors during setup, configuration, data taking and shutdown.

Authors

University of Wisconsin/Madison: A. dos Anjos
University of Bern: H.P. Beck
University of California, Irvine: S. Kolos
CERN: B. Gorini, M. Wiesmann

Document Change Record

Title:	ATLAS TDAQ Error Handling and Error Reporting in TDAQ Applications		
ID:	ATL-D-EN-0003		
Version	Issue	Date	Comment
0	1	12-March-2004	Error Reporting Working Group initial input.
0	2	31-March-2004	Adding homework
0	3	6-March-2004	few corrections added
0	4	5-May-2004	HP: Added feedback of combined developers meeting
0	5	27 July-2004	MW: Added feedback from Matthias and Serguei
0	6	28 July 2004	MW: Improved English
0	7	29 July 2004	MW: Large changes after discussion with Serguei, added some use cases.
0	8	31 August 2004	MW: Large changes after discussion with everybody: corrected typos, clarified sections, converted diagrams to UML.
0	9	7 October 2004	MW: Changes after discussions
1	0	18 December 2004	MW: Changes after Doris's comments

1 Introduction

The need of a global policy on the handling and reporting of errors has been identified by the working group for Error Handling and Fault Tolerance[1].

This document is an attempt to propose an implementation view on the realization of the above identified policy. It is therefore more specific in spelling out policies that affect the actual coding of TDAQ applications and other software components used by TDAQ applications. Furthermore, it aims for a common and simple to use approach throughout TDAQ:

- Commonality: if applications can behave as much as possible in the same way, it is easier to debug them and comprehend erratic behaviour;
- Simplicity: the less code to be maintained (e.g. in error critical areas), the easier the overall system can be understood, managed and used.

One has to keep in mind that at the current state of available components and applications, ad hoc solutions and policies for the handling and reporting of errors have already been defined by the various sub- and sub-subsystems in TDAQ. Although these individual solutions serve the purpose of every individual subsystem the forthcoming need for an integral and coherent policy becomes evident – especially, once fault tolerance procedure will need to be implemented.

It may not be an easy task to post-implement the here proposed policies into the existing code. However, it may still be possible to follow an evolutionary procedure in changing existing error handling and reporting in the current software components on a component by component basis. If this fails, an important rewriting of existing applications and components would be required at a non-negligible cost of human resources.

This document only describes the mechanism to collect errors and how to send them out of applications and libraries. Many other aspects of the handling of errors, like the actual interface of the system, the nature of the display, transport and storage mechanisms, the actual path taken by error once they have been reported (in particular in relation with the control tree), and the error aggregation mechanism (combining repeating or related errors into one comprehensive structure) are not described in this document. Those issues are very important and should be discussed in a separate document.

2 Glossary

Error	An event the prevent a component from performing its task.
Issue	An event that needs attention or handling. Errors, fatal or not, warning, information notification and debug messages are considered issues in this scope.
Synchronous Error	An error that is signaled in the context of a request or invocation (and concerns said request or invocation).
Asynchronous Error	An error that is signaled outside of the context of a request or invocation.
Transient Event	An event whose occurrence is time related.
Exception	A synchronous error that is signaled using the C++ exception mechanism
Error Code	An error condition that is encoded in a simple value, generally an integer.
Error Message	An informative message that signals and describes an error.
Debug Message	A message used for debugging purposes.
Debug Level	A level characterising the detail of debugging that is to be analysed. Low debugging levels mean that only coarse information is reported, while high debugging levels imply that more detailed information is reported.

3 Error Reporting Path

When a TDAQ application or library gets into an error condition two paths can be taken:

1. Fix the error, internally to the application, possibly issuing an informational message, or
2. Report the error condition and expect an external agent to handle the condition.

Both scenarios require a different programming approach and design techniques and yet can be caused by similar error conditions. It is sometimes unclear which of the above paths should be followed. In case of doubt, the application or library designer should consider at the point in code where the condition arises if the error can be solved locally. If the answer to this question is *'yes'*, path 1 should be taken, otherwise path 2 needs to be chosen. Also occasionally the answer to such a question may be *'yes, but not at this application level'*, in which case the error should be passed up and up to possibly several application layers until one is capable of solving the error or pursuing path 2.

If path 2 needs to be taken, the error stream goes outside the scope of the application. The error will hence be dispatched to an external component, for instance a log file, a reporting system, the standard error stream (stderr) or the expert system in the controller. Complimentary information need to be attached to the error, date, time and exact location of the error, textual description and additional parameters (typically defined by specialised subclasses). The error may then be analysed and possibly fixed, but the application itself has no control on this and simply waits for corrective actions to be taken from outside. Table 1 shows the matrix of the different handling policies, depending if the error is internal or not, and whether it can be solved or not.

	Can solve issue	Cannot Solve Issue
Internal interface (intra-application)	• Correct issue	• Throw exception
External interface (inter-application)	• Correct issue	• Report error

Table 1: Issue handling matrix

The objective of the Error Reporting System (ERS) is to provide a unified way of reporting issues (both errors and other types of issues) in the scope of an application (intra-application) as well as outside of that scope, (inter-application). For obvious reasons, in the first case, errors should be transmitted using the established programming techniques, i.e. return codes or exceptions. In the second case we need an abstract and simplified application programming interface (API) that is independent of the actual mechanism used to transport and dispatch issues. This way, developers need not to be concerned about the underlying strategies for message transport and display.

ERS can also be used inside a given application for error dispatching in multithreaded code. In such code, threads that encounter an error cannot report it internally using the exception throwing mechanism. In such cases, if a thread A encounters an error, it should put in into an ERS stream, and error handling thread B should read the error from the stream and handle it in the appropriate way (see 6.1.7)

The underlying transporting engine for the API can thus be chosen at will from other projects or built from scratch without changes to the API. This way, when handling an error, the developer can concentrate on the describing of the error and not to worry about the dispatching mechanisms, the code becomes clearer and less verbose and all error handling code is factored out.

4 Issue Content

When an issue is detected and the component that detected the issue can no longer pursue its normal activity then the issue is an error. The component that encountered the error should report this issue to a supervisor and cleanly terminate the current activity (function or method call, service request). Reporting in this scope can mean anything from sending the error to an error reporting system (ERS), a simple return code from a function call or the throwing of an exception.

If the called code decides to report an error to the caller code, either using error code or throwing exceptions, the calling code will need to decide what to do about the error in order to resolve it. Error handling is done inside the same application (intra-application handling). There are two possibilities: either the caller can solve the error or it cannot. In the first case the application can continue its normal activity. In the second case, the error prevent the application from fulfilling its task and the error should be reported further up in the call chain or to the controller system.

Ultimately, errors will be dispatched either to a human operator or to an expert system. Both will be responsible for trying to diagnose the overall situation and to take corrective action. From this point of view, an error should carry a maximum of semantic information, to help either the operator or the expert system. This information means the type of error, its scope (where it happened), the reason for the error (why) and other related information like responsibility, transience, how much state has changed. Of course, in some cases such information will not be available and the framework should support ‘unknown’ values. One example of semantic information can be found in the SNMP protocol [7], this system is widely used in network protocols like HTTP or BEEP 8].

When an error is solved, i.e. the service can be provided, it is not an error anymore, but it remains an issue. Issues that are not errors can also be handled using the ERS mechanism. For instance, if a file is not found, the issue can be solved automatically by finding the request file on another file system. Yet the fact that the file was not found is still an issue because it might indicate a mis-configuration. Because of this, the issue still must be handled and dispatched by the ERS system, for instance as a warning.

4.1 Issue type

This is the most important information about an issue: what type of issue it is. The type of the issue is reflected by the class type of the issue instance. The Issue hierarchy is described in details in 6.1.3. In addition to that, each issue type should include a textual description that explains in human understandable terms, what the issue is.

4.2 Issue severity

One of the most important aspect of an issue is its severity and is one that has to be extremely well specified. While other definitions have been proposed. Here we try to give an exact meaning of the different severity levels. The definition given here is meant to be used only in the **local scope**, that is an issue is an error, fatal or a warning in a given context. The severity of an issue is defined in regard to the context to a service request. A fatal issue is to be understood as fatal **regarding a given service request**, it might be considered as a warning at another level of the system. The service request associated with an issue can be implicit in the case of synchronous errors (it is the method call that caused the exception), or must be explicit in case of asychronous issues.

Deciding if an issue is a warning, an error or a fatal error is not an action that components should take. A component **cannot and should not** decide of the implications of a given failure outside of the scope of the request. Deciding the severity of an issue on the global scope is behond the scope of this document and is an operation that should be carried out by a specialised component, typically an expert system.

Fatal Fatal means that the component that issued the issue is unusable. No request or work unit will be processed until the component is recovered, either by resetting it, or by physically repairing it. A fatal issue implies that the issue is not transient and the cause of the issue is internal to the service. Further requests to the service will fail unless the component is somehow recovered. As Fatal errors imply that components get corrupted and unusable, fatal errors should (hopefully) be relatively rare.

Examples of fatal issues are hardware failures and components whose internal data structures are corrupted and need to be restarted. In the case of hard drive I/O, media failures are fatal issues (except if the hard drive is replaced, retrying will not solve the problem).

Error Error means that the component that issued the issue failed to process a request, an invocation or a work unit. The component is working and will accept further request, and if they are correct might process them.

	Examples of errors are request with illegal parameters (transmitting an histogram with a negative size), requests that cannot be fulfilled because of insufficient resources (out of memory errors), unidentified transient errors (segmentation fault). This should be the most common category.
Warning	A warning means that requests are processed correctly, but that there exist some condition that other applications should be aware of. The warning can concern the quality of the data produced by the component that issued the warning (for instance a low sampling rate), or describe a condition that might lead to an error or a fatal if not treated preventively (for instance, lack of disk space). A warning can only be transmitted via the ERS, and cannot be thrown or expressed using error codes. *Examples of warnings include the use of deprecated methods and libraries, corrupted data that could be corrected, degraded quality of service (for instance a low sampling rate), loss of precision, the use of low quality user interface mode (for instance black and white display instead of colour display). In general errors that were corrected should generate warnings.*
Information	Status information describing the behaviour of the system. For instance the fact that a failure occurred and was corrected. Information notifications cannot be compiled out or their display suppressed. Information can only be transmitted via the ERS, and cannot be thrown or expressed using error codes. *Examples of information include state transitions, loading and unloading of libraries and modules and in general messages that signal the general state of the system.*
Debug3	Debug message that appears at a high rate; e.g. with every I/O operation of an application. Debug messages can only be transmitted via the ERS, and cannot be thrown or expressed using error codes.
Debug2	Debug message that appears at reduced rate; e.g. with every full treatment of an event. Debug messages can only be transmitted via the ERS, and cannot be thrown or expressed using error codes.
Debug1	Debug message that appears at well even more reduced rate; e.g. with every command from the run-control received.
Debug0	Debug message that appears only few times; e.g. successfully read configuration database file

The display an dispatch of debug message can be configured. Debug messages can be suppressed for certain debug level (e.g. 0-3) or for certain packages, or compiled out altogether.

4.3 Issue context

The issue context describes where and when the issue occured. Most of this information should be filled in automatically.

File	The file containing the code where the issue occurred.
Line	The line in the code where the issue occurred.
Package	The module of code where the issue occurred.
Time	The time and date when the issue occurred.
Responsibility	What component is responsible for the issue. We distinguish cases when the requester is responsible (typically the component was requested to perform an impossible or illegal

	action), and case when the component failed to performs the service it is supposed to perform.
Cause	Was the current issue caused by another issue. Typically a low level issue might result in a high-level issue. This field permits issue chaining – issues would be handled like in linked list fashion.
Host	The name of the host where the issue occured.
Process / Thread	The identity of the process and thread where the issue occurred.

4.4 Additional Information

Each type of issue will define its own additional fields to carry complementary information. Some general information fields will also help the recovery system or the operator. The use of the those fields is optional and by default, the ERS system will insert default values into them. Those fields are not used by the ERS system, but are meant to be used by the destination of the issues.

Textual Description	A human readable explanation of the issue, as complete as possible. This description should either be complete, or give a hyperlink to a complete description of the issue.
Completed	Did the request that caused the issue complete (no / unknown, default: unknown). The unknown value is important in the case of remote errors, where it is not always possible to determine if an action has taken place on a remote machine.
Transience	Is the issue transient, i.e is its occurence time dependant. Transient errors might no occur if the same request is sent later on. Thus it makes sens to repeat requests that yielded transient errors. Out of memory errors and deadlocks are typically transient issues (boolean, default: unknown).
Service type	What kind of service did the issue arise in: hardware, real-time service operating system (default: unknown). This information would typically be useful for a human operator, which would call different experts depending on the nature of the problem.
Responsible	What component is responsible to handle the issue (default: none).
Recover	Did the issue occur while trying to recover another issue? (default: unknown).

4.5 Examples

This section gives a few examples of issues and the different attributes that are associated with them.

4.5.1 Segmentation fault

While serving a request, a server process does a segmentation fault.

Severity	Fatal	The requested service was not performed, and the service cannot perform any further service until some external action (restarting the service) is done.
Transient	Unknown	
Responsibility	Server	
Completed	Maybe	

4.5.2 File not Found

A component is requested to load file X, but this file does not exist.

Severity	Error	The requested service (loading a file) was not performed, but the component is still working (hence this is not a fatal error).
Transient	No	Requesting the same file later will not work without some external intervention.
Responsibility	Client	The client requested a file that does not exist.
Completed	No	

4.5.3 Event not Found

A component, for instance a ROS, is requested event X, but this event does not exist.

Severity	Error	The requested service (sending an event) was not performed.
Transient	Yes	Requesting the same event later might work.
Responsibility	Unknown	The event might not exist, or the servier might not have received it yet.
Completed	No	

4.5.4 Incomplete Event

A component sends a event that is not complete.

Severity	Warning	The requested service (sending an event) was performed.
Transient	Yes	Requesting the same might yield a complete event
Responsibility	Unknown	
Completed	No	

4.5.5 Missing configuration

When asked to configuring itself using file X, a component did not find the configuration file, and fell back to a default configuration.

Severity	Warning	The requested service (configuration) was performed.
Transient	No	Requesting the same service will yield the same result
Responsibility	Client	File X does not exist.
Completed	Yes	
Cause	File Not Found	This issue was caused by another issue (file X not found), but was handled and changed into a warning.

4.5.6 Configuration failed

When configuring itself, a component fails doing integrity tests.

Severity	Fatal	The requested service (configuration) was performed and the service is not usable anymore.
Transient	No	Requesting the same service will yield the same result
Responsibility	Server	
Completed	No	

5 System Classification

In order to simplify the analysis and design process we classified the system into layers of increasing complexity, where possibly more refined methods for error reporting might be necessary. In order to classify a package a couple of parameters have to be taken into consideration. The first is dependence. Low-level packages depend on few or no other packages at all and are in direct contact with the OS facilities, for instance. The second is usability. Application packages, for example, are not meant to be used by any other software.

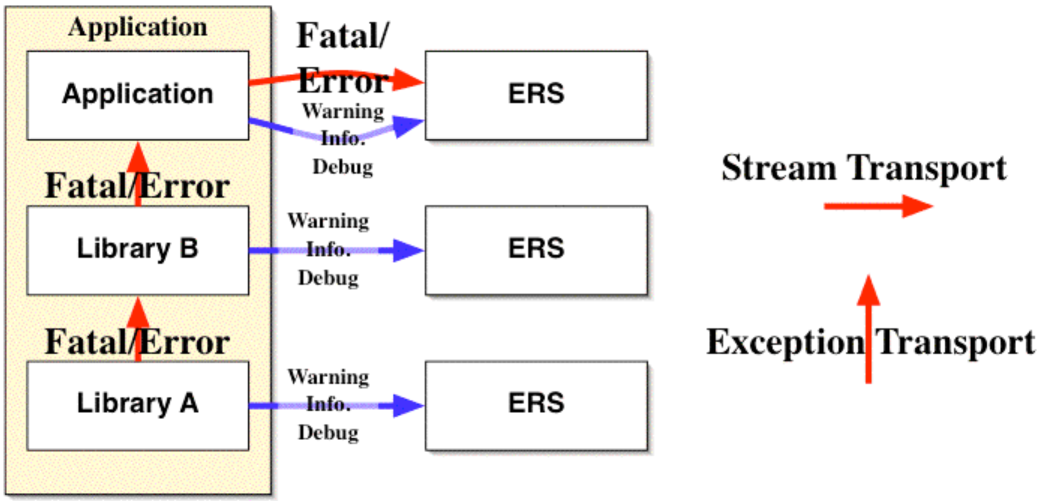

Figure 1: Layer interactions in ERS

Figure 1 shows the general interaction between layers in ERS. When issues arise in a component, they are either sent to the calling component, or to the ERS. Within an application, synchronous issues (fatal and errors) are transmitted using the exception mechanism. If a failure occurs inside a method or a function call, fatal errors and errors are notified using exceptions. Warnings, information and debug notifications are always sent using the ERS system. If the issue arises at the application level, for instance if a server encounters an error, it is dispatched using ERS.

For clarity's sake, asynchronous error handling within a multi-threaded application – which would be done via ERS – is not depicted. Conceptually, an error within a thread can be seen as an application error (it just happens that many application share the same memory space).

5.1 Libraries

Libraries are the building blocks of the TDAQ system. They include low level libraries and high-level middleware. Low level libraries are usually simpler and more robust, being simpler and having more specialised functionality and being used more intensively. Middleware libraries offer more advanced services like database access and remote method invocation.

Low-level libraries and middleware are both libraries used by the application and share the same interaction mechanisms, method are called, and in case of error, signal those using either error codes or exceptions. The main difference is that low level libraries cannot, because of space or performance limitations, rely on the same functionality than middleware. Thus while both low-level libraries and middleware can report warnings using ERS, they cannot use the same ERS bindings. While it is reasonable for a middleware to report issue using a distributed error transport system like the Message Reporting System (MRS), this is not reasonable for a library that is loaded into an embedded system.

In particular, in the cases of low-level libraries used to implement high-level ERS bindings cannot be used do dispatch their own issues. For instance, if an error occurs within a library used by MRS, the errors cannot be sent to the MRS biding because of circularity issues.

This means that all libraries will share the same ERS **interface**, but will probably link against very different **implementations**. Thus the interface should be designed to be as versatile as possible and accommodate both low and high-level implementations. The ERS interface should thus appear as a low level library, but be able to link against high-level middleware implementations. The different ERS bindings are discussed in Section 6.

In practice libraries should take the following actions in the advent of issues.

- Catch errors from the lower-level libraries or the operating system as soon as possible.
- Send Warning and Information message to ERS.
- Send Debug messages to ERS.
- Report errors to the calling layer using either error codes or by throwing exceptions.

5.2 Application Layer

This encompasses the working packages that build, on the top of the two precedent layers. Applications, as components of the TDAQ system, are supposed to integrate the overall error reporting system and inject error messages and status information into it.

Working packages at this level do:

- Catch errors from the low-level libraries and middleware
- Send messages for debugging purposes
- Send informative or non-fatal error messages using ERS.

Synchronous inter-application invocations, like RPC services, Java RMI or CORBA synchronous calls are a special case of inter-applications communication. While they are inter-application communication, they offer the same mechanism and context than a local function call, and should thus use the same error handling than local method calls. So if an error occurs during a synchronous CORBA invocation (not one-way calls), this error is dispatched using the CORBA exception mechanism. This in turn implies that ERS offers mechanisms to interact with CORBA exceptions.

5.3 Application Control

Applications in the TDAQ system are managed by a special component, called a controller. This controller is responsible for managing the state of the applications. One aspect of this task is error management and recovery [4]. Because of this, the controller must be able to receive ERS messages from the controlled applications. Other destination for the error stream include debugging programs, logs, operator interfaces.

While there is both overlap and interactions between the controller design and the error handling framework, describing how the controller handles issues is beyond the scope of the present paper.

6 Architectural View

In order for the ERS system to be available at all parts of the system, the interface should be placed at the lowest level and have no or little dependencies. The API should be made as independent as possible from implementations. The actual implementations (like transporting the notification over some middleware) should be hidden from the user with possibilities to chose the actual implementations and its behaviour at compile, link or run time.

The default implementation of the ERS interface should be designed to be both light and self sufficient, thus enabling developers to benefit from the API with little configuration and no dependencies (compiles out of the box). The use of more advanced implementations might need some configuration and adaptation. This should be simplified as much as possible, to avoid common errors and to encourage usage.

The package should offer comprehensive tools to help adding handling code to existing software.

An initial high level design is sketched in the following sections.

6.1 High-Level Design

The TDAQ ERS is composed of two different mechanisms that work cooperatively to satisfy all the requirements of this subsystem. One being the error reporting mechanism that sends debug and error messages out of the application scope using an error stream, and the other being the reporting of error states between the various layers within applications using error codes and exceptions.

6.1.1 Architectural considerations

One of the goals of the ERS package is to be used in a pervasive way in the TDAQ system. This means the framework has to adapt to different working conditions, the API should be usable for small, standalone programs with little to no external dependencies, but also work when linked against the online configuration database. It should be possible to compile out debug messages at different levels, and configure the implementation and behaviour of the ERS stream at compile time, link-time, or run-time. Run-time configuration would typically be extracted either from the configuration database, or in standalone mode, from the environnement variables.

In the simplest case, the programmer could simply use a default implementation of the ERS stream and have the behaviour automatically determined by the linking type (i.e send errors to stderr in standalone, to MRS when linked against online software). More avanced programems could open multiple ERS stream for different types of semantic issues (for instance a stream for general issues, and another for timing/performance issues) and internal stream (to dispatch errors from worker thread to the main thread).

6.1.2 Issues

The error reporting mechanism itself is depicted in Figure 2. An issue can either be thrown as a C++ exception (if it is an error or a fatal error) or sent via a single ERS stream (it can be either an asynchronous error or fatal, a warning, an information or a debug message). An issue is represented by the same object, regardless of the fact that it will be thrown as an exception or sent as a message.

The actual implementation of the ERS stream is not visible to the developer and can handle issues in different ways. Implementation could send the error to the standard error output (*std::cerr*), write it into a log file, transmit it using a system like MRS or even send an e-mail. In all cases the stream interface remains the same. Different components can be configured to use different implementation of the ERS stream.

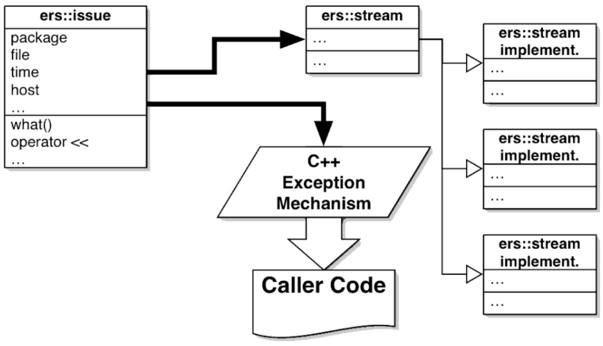

Figure 2 Error dispatching pattern – arrows represent the flow or errors

There are a number of fields that should be present in an issue object in order to allow the catcher to make full use of the exception or report the error if it cannot handle the situation. Every issue contains the fields described in Section 4 as well as a short human comprehensible text description. As much as possible, this information is filled in automatically. The method `what()` should report a human-readable format of the issue. Specializations of the exceptions can define specific fields but the overall format should be the same and controlled by the implementation of the standard exception class, except where new information is added. This proceeding improves message understanding and manipulation likewise with the base error object shown above. Therefore, the base exception class shall be designed with all those parameters in mind, as shown at Figure 2. This sketch also introduces the possibility to generate standard messages from the exception, in the case the application or library wish to communicate the occurrence of such an exceptional condition to the system operator.

6.1.3 Issue Tree

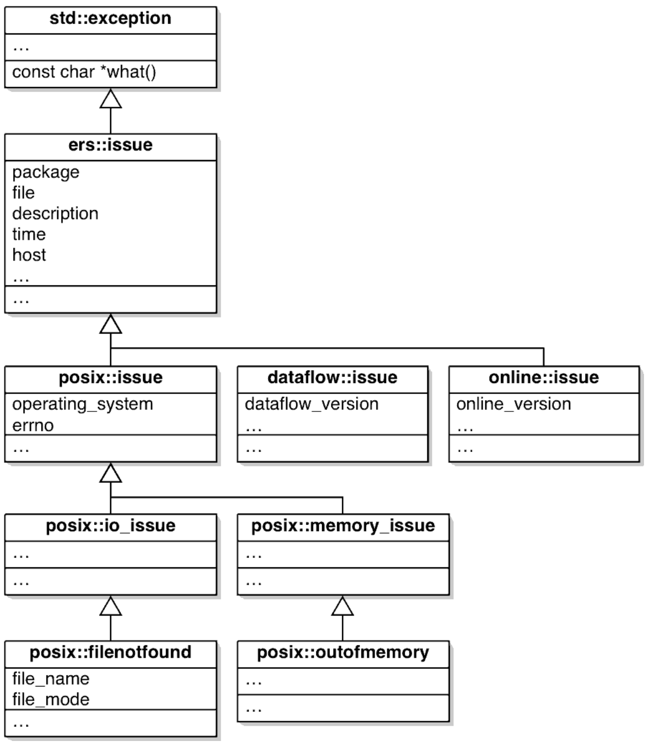

Figure 3 Error Inheritance Tree – arrows represent inheritance relationship

In order to keep the system organized, ERS working group proposes to aggregate all errors that may be generated during execution in a tree, described in Figure 3. The topmost type is *std::exception*. Thus all 'issue' objects are subclasses of C++ exceptions[1]. The subclass *ers::issue* will define all the fields described in the current document except the textual description (which is defined by the what method in the C++ exception class). The severity of an issue (fatal, error, warning, etc…) is an attribute of the *ers::issue* object. This way, an error can be transformed into a warning, or a debug message[2]. Below this *ers::issue* object there are detector specific sub-trees and one tree with all the common errors, like for instance POSIX-related issues. While C++ permits multiple inheritance, this should be avoided in the issue tree, to avoid confusion in the exception handling code.

This allows each application level to focus on the error levels it can cope with. exemplifies this layout. The arborescence can either be made more complex if more ATLAS subsystems decide to adopt a common ERS (without changing the ERS API) or simpler, if package maintainers do not wish to specialize too much their package exception tree. Common errors, like file and memory errors should be grouped into a common sub-tree.

6.1.4 Issue Administration

Particular care must be taken to handle the issue tree, so that there is clear responsibility and documentation of all instances of the issue. All issues definition should be managed in some central repository. Beside the common issue tree, detector and software package specific trees of issues would be designed. As much as possible, developers should use issue objects defined in the common tree (i.e. not redefine file-not-found). Restrictions for the issue tree might be considered as the project evolves and programmers get experienced with exception handling on their packages, typically by removing vague issues and replacing them with more precise and informative types.

[1] The C++ language permits the throwing of any object as an exception. This means that std::exception is no particular object. For clarity's sake, the issue class inherits from the std::exception class, even for issues that will never be thrown (information messages). This has no implication as the only property of the std::exception is the what method that gives a textual description.

[2] Whenever the serverity attribute is interal to the object or given by the context / stream it is in is still a point to be cleared.

6.1.5 ERS Stream

The core of the issue dispatching are the exception mechanism and the ERS stream. The fist element is given by the C++ language, the second needs to be implemented and is described here. The core idea of the ERS stream is to offer a way to transparently send issues to different destinations. ERS streams are used as normal C++ streams, simply they offer tools to streams issue objects.

The ERS stream interface hides different ERS implementations. Different implementations can be bound to the code at different points in time, at compile time, and link time, and at runtime. Those different linking options reflect different needs and usages for the package. Because the implementation of the `ErrorStream` is hidden to the user, it can be bound dynamically to both the current terminal error and output streams or to a specialized messaging services. Figure 4 presents possible implementations: null-macros, printing out on std::err, writing into a log file, transmission via the Message Reporting Service (MRS). Additionally each streams has an associated filter, depending on this filter, debug messages of a certain level will be filtered out.

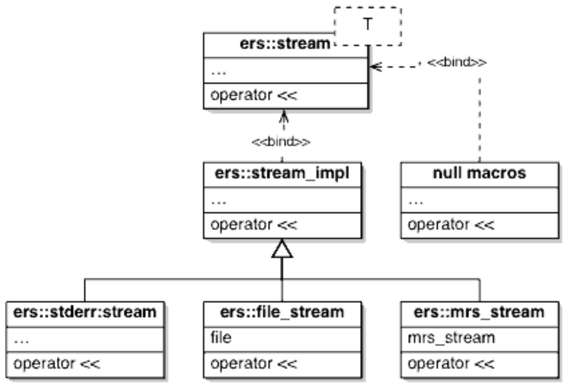

Figure 4: ERS stream implementations

Special implementations of the `ErrorStream` may allow filtering and treatment of the output messages at the source or remote agent (using the same API). All such functionalities are hidden from the user perspective. This makes using the ERS simple and effective. Filtering of messages, in particular of debug messages can be implemented by different ERS implementations and selected at different times (compile, link, run). In particular, it should be possible to compile out code handling debug messages. This allows for a configurable trade-off between flexible management and performance.

Beside a default ERS stream, the user should be able to use an arbitrary number of ERS streams linked with different implementations. This way a programmer can send different debugging messages to different destinations during development, and filter some out in the final system while dispatching the others to a remote control system. Programming macros may be implemented to address the correct use of message reporting in the ERS. Apart from that, debugging macros will be provided. Both compilation and runtime options shall exist to turn on and off debug output. These macros directly contact the system error stream to send debug messages to the appropriate place. Dynamic disabling happens through stream filtering as explained before.

6.1.6 ERS Reporting

The messages reported by the ERS should, as much as possible, try to indicate the cause of the issue they report and **not** their consequences. For instance, if a database fails to load, the caller is more interested in why the database could not be loaded instead of the implications of this failure. Thus information about the error itself, not its consequences should be inserted into error structure with first priority. The implementer of a given component should concentrate on detailing the error at hand, not try to second guess the impact on the overall system.

6.1.7 Inter-thread usage

One of the usage of ERS threads if for doing error handling in multi-threaded applications. In such settings, exceptions thrown within a thread have to be handled within the thread. In order to centralize all exception handling in one location of the code, a ERS stream can be used. Figure 5 illustrates this type of usage. The application has three worker threads (A, B and C) and one error handling thread, that is responsible for control and error management. When one of the worker threads encounters any issue (fatal error, error or warning) it sends it to the internal ers::stream. The error handling threads reads this stream (typically a blocking read) and receives the issues objects and can handle them as needed – this is basically an event loop, where the events are the exceptions raised by the different threads. In this case, the ERS stream is not connected to any entity outside the application, and is only visible internally. Of course, the main thread can forward the issues to another ERS stream.

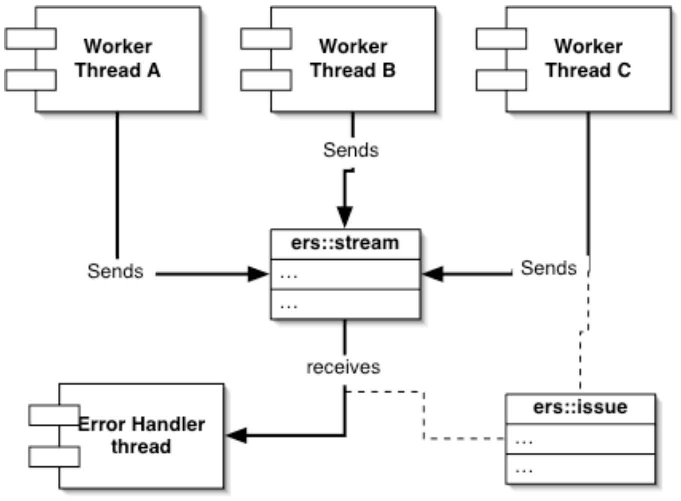

Figure 5: Inter-thread communication with ERS

The thread framework can be modified so that for each thread, uncaught exceptions are send to the error handling thread (instead of aborting the whole program).

6.1.8 Exceptions

An exception marks the occurrence in the program where the current framework cannot complete, by any other means, its current activity. Exceptions represent a way to present irreversible error conditions to the user of a method.

When an exception occurs in C++ specifically, an operation called stack-unwinding is launched. This operation causes all objects allocated on the stack to be destroyed. If an exception occurs at this time, the running code immediately aborts. Thus exceptions should **not** be used in destructors.

Except when thrown, exceptions induce little overhead to a program, but stack-unwinding has some price. Because of this, exceptions should not be used as mean for normal control flow, like for instance to replace non exceptional returns. On no account should they be used to replace goto's. It is interesting to note that the possibility of throwing an exception may introduce new paths to the program execution, since a call may prematurely return when a particular exceptional condition is met. One should be advised to catch exceptions as soon as possible to avoid unpredictable application behaviour and introduce complex possibilities to system execution.

Error values should slowly be abandoned in favour of exceptions in TDAQ. The migration should be smooth and allows re-thinking if there seems to exist areas where error codes are shown to be more appropriate than exceptions. For more information on exception programming, please refer to the appendix A1.

6.1.9 Error Codes

Even if programmers are encouraged to move into using exceptions to report abnormal program conditions, there may be cases where the use of error codes is justified. In those cases, the developer should indicate why using error codes is more appropriate and determine a set of actions which should be taken when an error code is set.

There are two main advantages at using exceptions instead of error codes. The first is convenience: exception can contain more information that numerical error codes, and object polymorphism makes error handling more easy. Still all functionality offered by exceptions can be implemented by hand, information can be stored in thread-local variables and user-defined structures. The second reason is quite simple. Exceptions cannot be ignored. This ensures that no error condition notification is lost simply because one method call does no checking on its return status.

7 Use Cases

This section describes some use cases for error handling code.

7.1 Sending an exception to a stream

A method returns an exception, but the calling code transforms it into a warning and sends it to the ERS stream.

```
try {
     data = updateData();
} catch {DataUpdateFailedException &e) {
     ers::warning << e ;
}
```

7.2 Precondition violated

The programmer adds some precondition check in the begining of a function. In debug mode, this leads to an exception being thrown. In final mode, the precondition code is compiled out to maximise performance. Preconditions violations are marked as being errors with the client being responsible and no state change being done.

```
double square_root(double  x) {
ers::precondition(x  >= 0,"negative square root requested");
…
```

7.3 Postcondition violated

The programmer adds some postcondition check in the end of a function. In debug mode, this leads to an exception being thrown. In final mode, the precondition code is compiled out to maximise performance.

Preconditions violations are marked as being errors with the server being responsible. Whenever there was a state change is passed as an additional parameter, in this case, false.

```
double square_root(double  x) {
double result ;
…
ers::postcondition(result  >= 0,”negative result for square root”);
if (x < 1.0) ers::postcondition(result > x,” x > sqr(x) when x < 1”,false);
if (x > 1.0) ers::postcondition(result<x, “ x < sqr(x), when x > 1”,false);
return result ;
} // square_root
```

7.4 Assertions

Code uses interal assertions. In debug mode, an exception is thrown, in final mode, the assertion is commented out. Because assertion contain little semantic information, they should only be used for sanity checks.

```
double sum = 0 ;
for (i=0;i<max;i++) ;
{
     ers::assertion(i<max,” out of bound index”);
     sum += value[i] ;
}
```

7.5 Debugging Messages

Debugging messages are sent to the ERS stream, the messages can be simple strings with a severity qualifier, debugMessage objets, or any issue.

```
…
ers::debug << ers::debugLevel(1) << “file  open” ;
ers::debug << ers::debugMessage(3,“database  loaded sucessfully”);
try {
     …
} catch(DemagnetizerOffline &e) {
     ers::debug << ers::debugLevel(1) << e ;
}
…
```

7.6 Aborting with an issue object

An issue is deemed fatal. The issue object provides an abort() method that stops execution, displays the issue information and dumps a core.

```
try {
     do_somthing();
} catch (OutofCheese &e) {
     e.severity = ers::FATAL ;
     e.abort();
} // try / catch
```

7.7 Error Handling thread

A thread registers to listen to a ers::stream while other worker threads dispatch their exceptions to this thread.

```
void error_handling_thread(void  *ptr) {
     ers::stream *s = (ers::stream *) ptr ;
     while(true) {
          ers::issue next ;
          *s >> next ;
          … // handle issue
     } // while
} // error_handling_thread

void worker_thread(void  *ptr) {
     ers::stream *s = (ers::stream *) ptr ;
     try {
          do_thread_work();
     } catch (ers::issue &e) {
          *s << e &
     } // try / catch
} // worker_thread
```

7.8 Legacy functions

Legacy C code, like for instance the POSIX API, do not throw exceptions. Instead error conditions are signaled using return code and thread local variable errno. For such API utility function that build up the correct issue object can be build. For instance the common part of the framework should include classes and utitilty functions for the POSIX API. In such cases, it makes sense tu use the *factory* design pattern, that is a class method that builds the correct concrete example. Such factory methods would typically take the same parameters than the POSIX call that failed.

The following example shows the failure of the POSIX open call with the code that builds the correct issue object from the parameters and the value of errno and throws it as an exception. The actual issue objet type will depend on the error described by the errno variable, but will be a subclass of posix::exception and can thus be caught and queried as such.

```
void * map_memory(const char* path) {
     const int fd = ::open(path,O_RDONLY, 0);
     if (fd<0) throw posix::factory::open(fd,O_RDONLY,0);
     void *ptr = ::mmap(0,DATA_SIZE,PROT_READ,MAP_FILE,fd,0) ;
     if (! ptr) throw posix::factory::mmap(0,DATA_SIZE,PROT_READ,MAP_FILE,fd,0);
     return ptr ;
} // map_memory
…
try {
     map_memory(filename);
} catch (posix::file_not_found  &e) {
     … // handle file not found – can only created posix::factory::open
} catch (posix::invalid_access  &e) {
     … //  handle invalid access
     … //  can be created by both  posix::factory::open and  posix::factory::open
} catch (posix::issue &e) {
     … //  default handler for posix issues
}
```

For common POSIX objects like semaphores, it makes sense to wrap them in classes whose methods would throw exceptions in case of error. In a general way, exceptions should be thrown by the common framework.

8 References

1 Proposal from the Connect Forum for the formation of a dedicated TDAQ Error Handling Policy working group

2 Lassi Tuura, *Exception Handling Guidelines for C++*

3 Bjarn Stroustrup, *The C++ Programming Language*, 3rd ed. 1997

4 A. Kazarow, G. Lehmannn, D. Liko, H Zobernig, S. Wheeler *ATLAS TDAQ Controller Requirement* ATL-DQ-ES-0054

5 *Atlas C++ Coding standard specification* ATL-SOFT-2002-001

6 Marshall Cline *C++ FAQ Lite* *http://www.fmi.uni-konstanz.de/~kuehl/c++-faq/exceptions.html#faq-17.3*

7 Jonathan B. Postel *Simple Mail Transport Protocol* Internet RFC 821 Appendix E, theory of return codes. *http://www.faqs.org/rfcs/rfc821.html*

8 M. Rose *The Blocks Extensible Exchange Protocol Core* Internet RFC 3080 http://www.ietf.org/rfc/rfc3080.txt

A1 Programming with Exceptions

Exceptions

Exceptions are one possible way to report errors within layers of an application. It is perfectly appropriate for reporting synchronous errors since they can't be ignored and will result in terminating the thread and eventually the process if no error handler catches it. This is a clear difference with error codes, where return values can simply be ignored. Exceptions are objects and can thus carry additional information that simply error codes. While additionally information can be attached to return values using thread-local variables, this is more complex and error prone.

Exceptions are well adapted for signalling synchronous errors, that is error that occur during the execution of a request. They are not usable for asynchronous errors (errors that are not related to request / reply context), nor for the transmission of informational data (for instance warnings). Thus, a second transmission mechanism is needed for both asynchronous errors. Asynchronous error can occur when an error occurs in a detached thread. In this case, the main thread might listen to the ERS stream of all workers threads and dispatch errors as they occur.

Error recovery can either be internal to the application in which case the exception is caught and the issue handled using C++ code, or external in which case it is transmitted either to either the controller or a human operator that will try to resolve the issue by sending configuration and state change commands to the application.

Destructor

Exceptions can be thrown by constructors, functions and methods but never from within a destructor. The issue with exceptions thrown from within a destructor is that it breaks the stack unwinding process. Thus, if an exception is thrown from within a destructor during stack unwinding, the *std::terminate()* method is called, which aborts execution of the process. Thus, exception must never be used within destructors.

catch argument type

The catch should always receive a reference on the exception. This is because the thrown class may be a derived class with virtual methods to access its specific fields. If the exception is passed by value, the object is copied into the base class and all information specific to the original exception object is lost.{

```
try {
  …
}
catch( exception& e )
{
}
```

Exception hierarchy

When function of a package or method of classes throw different specific exceptions, it is recommended to provide a common base class to allow the calling code to catch all exceptions related to a package. This way, new exception types can be added without forcing the calling code to change its exception catching code, if the code catches the general base class exception.

Exceptions can be any C++ data type, including non objects. They can also use multiple inheritance, but such practices are not recommended.

Exception Safeness

When a method or a function calls another method or function that throws an exception, the call stack is unwound. This means that all variables allocated on the stack are properly destroyed and calls aborted until an appropriate exception handler is met. This thus automatically cleans up most data in terminated function or method calls. But it is not enough.

For instance, if the function allocates an object on the heap by using the *new* instruction and assign the address to a standard C or C++ pointer (*obj**), the object wont be destroyed when the function is terminated by an exception. And this will result in a memory leak. Another example is with resources which have been initialized and need to be de-initialized before the method or function terminates. This requires special care to make the code exception safe in the sense that the system state is properly restored in a clean state.

The first strategy is to use an exception handler. Consider the following code example which is not exception safe.

```
void foo {
initialization();
do_something();
de_initialization();
}
```

If an exception occurs in the *do_something()* method, it will short cut the *de_initialization()* call and the system is not properly restored. Mutex locks or semaphore are a good example. An exception safe way to implement this method would then be the following.

```
foo()
{
initialization();
try
{
  do_something();
}
catch(...)
{
  de_initialization();
  throw;
}
de_initialization();
}
```

But there is also a better, simpler and cleaner way to ensure exception safeness. This requires that the resource is initialized and de-initialized by a constructor and destructor, which is known as *resource acquisition is initialization* []. This thus requires an appropriate design of the resource to manage which is not always possible. But it should be used wherever it can be used. This requires to provide a class that manages the resource.

```
class Resource
{
public:
Resource() { initialize(); }
~Resource() { de_initialize(); }
};
```

The user exception safe code then becomes the following.

```
foo()
{
Resource r;
do_something();
}
```

As one can see the code is simpler than with the first solution. It is also much less error prone since the user has only to take care to allocate the resource it want to use. A resource may be a network connection, a file, a mutex lock or anything you're application may need.

For dynamically allocated object, the same behaviour is obtained by using smart pointer. A smart pointer has exactly the same behaviour and interface than a normal pointer except that it has a constructor and destructor. Smart pointer uses an underlying reference counting system so that when the last smart pointer to an object is destroyed the referenced object is destroyed.

This behaviour is totally transparent for the user and ensures exception safe code. Reference counting introduces a slight performance overhead so that in performance critical applications some more optimal strategies must be used. Use of smart pointer needs some care to avoid circular references which may result in memory leak.

Constructor

Exceptions can be safely used in constructors. C++ will take care to call the appropriate destructor of partially constructed class and to delete allocated space by the new instruction. If the constructor of class X has a sequence of initialize instruction, and an exception is thrown in one of them, C++ will call the destructor of the already initialized variables in reverse order.

```
X::X() : a(...), b(...), c(...), d(...)
{
}
```

If an exception occurs in *c* then the destructor of *b* and *a* will be called in that order. Inside the constructor method body you should use classical exception safeness strategy. Note that *a*, *b*, *c* and *d* should be the resources presented in the previous section.

A2 Error Handler complexity

There might be a large variety of Error handler implementations according to their complexity or the complexity of error they may have to handle. At a very low level one may find very simple error handlers as shown the in the following example:

```
if( ! ptr)
throw std::exception( “ptr detected to be NULL” );
```

Error detection and decision are combined in the ‘if’ instruction and the action performed is to throw an exception which is equivalent to a fatal error handling. Note that the error is fatal from the point of view of the method. The higher level error handler that will catch this exception may take some actions to recover from this fatal error, or report it via the ERS.

Here is a simple example of recoverable error handling with active waiting:

```
do
{
fd = ::connect( hostAddr );
if (fd>=0) break ;
report( “Failed connecting, wait 2 seconds” );
sleep( 2 );  // should be configurable
} while( fd < 0 )
```

A higher level error handler will catch the errors reported by lower level error handler. The C++ language provides the *try catch* clause syntax to define the decision to take and associated actions.

```
int fd = ::connect( hostAddr );
try
{
// code that may throw an exception
}
catch( OpenDatabaseFailedEx e )
{
// handle OpenDatabaseFailedEx
}
catch( std:exception e )
{
// handle standard exceptions
}
catch( ... )
{
// handle any exception
}
```