ATLAS

ATLAS TDAQ

Error Reporting System Architectural Analysis and Design

Document Version:	1.0
Document ID:	ATLAS-TDAQ-2002-XXX
Document Date:	31/01/2005
Document Status:	Draft

Abstract

This document describes the design for the Error Reporting System (ERS).

Institutes and Authors:

CERN: Matthias Wiesmann

Table 1 Document Change Record

Title:	ATLAS TDAQ Error Reporting System Architectural Analysis and Design		
ID:	ATLAS-TDAQ-2002-XXX		
Version	**Issue**	**Date**	**Comment**
1	1	22-Apr-05	Initial version

1 Introduction

The Error Reporting System offers an unified system for handling error, warning and debug messages in the TDAQ service. This system is designed to be used both internally by TDAQ applications and to communication between TDAQ applications.

1.1 Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made about the system.

1.2 Scope

This document describes the general Error Reporting System architecture along with the client interface (that is sender of error, warning and logs), it defines the interface for transport mechanisms but does not describe the implementation of the error transport systems.

1.3 Glossary

Asynchronous:	An error is asynchronous if the notification of the error occurs outside of the context of the request that caused the error. A server that sends a message to signal an error signals an asynchronous error.
Debug Message:	A message used for debugging of an application.
Error Code:	An error condition that is encoded in a simple value, generally an integer.
Error:	An event that prevents a component from fulfilling its task.
Exception:	A synchronous error that is signalled using the C++ or Java exception mechanism.
Fatal Error:	An error that signals that a component will not be able to perform its task in the future without external intervention.
Issue:	An event that requires attention, issues can be errors, fatal errors, warnings and debug messages are issues.
Synchronous:	An error is synchronous if the notification of the error occurs in the context of the request that caused the error. A function that returns an error code signal a synchronous error.
Transience	An event is said to be transient if its occurrence is time dependant. A transient error can occur or not depending on the timing.

1.3.1 Acronyms and Abbreviations

CERN	European Laboratory for Particle Physics
DTD	Document Type Definition. A document that contains the description of the structure of some XML based format.
ERS	Error reporting System
Log4cxx:	A open-source framework to report and transport logging messages. This project is maintained by the Apache Foundation.
MRS:	Message Reporting System. A TDAQ component responsible for the transport of errors, warnings, and information messages.
POSIX	Portable Operating System Interface. A standard for operating system APIs.
URL	Uniform Resource Locator. A uniform way to designate resources.
XML	eXtensible Mark up Language. A generic, text based format used to store and transmit information.

1.4 References

[1] Error Handling and Error Reporting in TDAQ Applications, A. dos Anjos, H.P Beck, S. Kolos, M. Wiesmann

[2] *Modern C++ Design*, A Alexandrescu Addison-Wesley 2002

[3] *Xerces Library http://xml.apache.org/xerces-c/*

1.5 Overview

This document is structured in the following way: Section 2 contains the architectural goals and constraints, Section 3 contains the logical view and Section 4 presents the use cases, Section 5 shows the process view of the system and Section 6 presents the implementation.

2 Architectural Goals and Constraints

The goal of the ERS system is to offer a simple, lightweight system to handle both errors and warning, log and information messages. The framework should be compatible with existing error handling systems (like C++ exceptions) and coding practices. And in general fulfil the requirements presented in [1].

2.1 Compatibility

2.1.1 Standard Exceptions

The ERS should be compatible with language dependent exceptions, in particular C++ exceptions. It should be possible to ERS issues as standard STL exceptions, but also to construct ERS exceptions out of STL exceptions.

2.1.2 Error Streams

The ERS should be compatible with Unix error streams. That is it should offer facilities that make it possible to:

- Send ERS issues to the standard error stream
- Offer interfaces similar to the standard error stream to stream to the ERS system.

2.1.3 Existing Exceptions

The ERS should be compatible with existing exception classes that are used in the current system. This includes:

- The dataflow `DFError` class
- CORBA exceptions

The ERS should offer similar interfaces to those classes, and accept instances of such classes as initialisation parameters for instantiating ERS Issues.

2.2 Dependencies

The goal of the ERS is to adopted system wide and to be used at all stages of software development. Because of this usage of ERS must be simple and dependencies to other software packages should be minimal.

Because of this, the ERS system should clearly decouple requirements for the core system and requirements for the transport backend. Dependencies for other frameworks should be limited to bare bone functionality, like threads or XML parsing.

2.3 Performance

As error checking implicitly implies checks and additional branches in the code. In order to be used widely the Error Reporting System should not incur a serious performance penalty. This can be achieved by several techniques techniques.

- Compile time checks – such checks permit better debugging and can be compiled out during compilation.
- Provide facilities to disable error reporting facilities either at compile time, or at runtime.
- Implement filtering as early as possible to minimise traffic.

2.4 Serialization

One important feature of this package is the ability to transmit Issues between processes, to store and retrieve them on stable storage and in general to serialize and un-serialize them dynamically. This feature is easy to achieve in Java, but requires some thought in order to be implemented in the C++ language that does not support dynamic object building directly. This can be addresses using the factory pattern [2].

3 Logical View

The ERS can be used both internally and externally.

3.1 Overview

The ERS can be decomposed into two logical elements: Issues and Streams. Issues are the objects that represents the different problems the system handles, Streams represent channels where Issue are transported. In this sense, Issues are both messages send into stream and exceptions that can be thrown and caught. Figure 1 illustrates the general structure of the system.

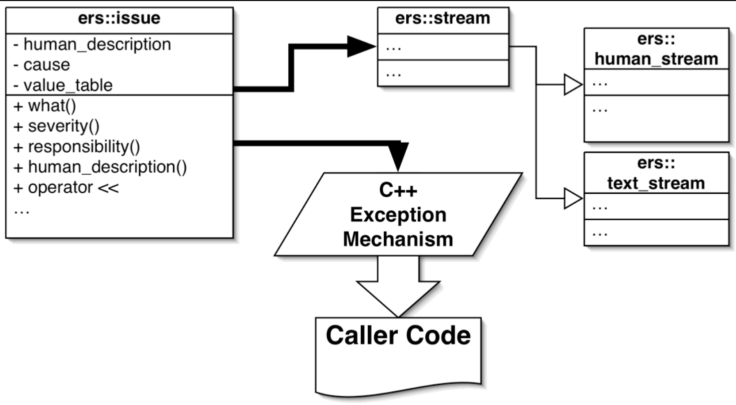

Figure 1: Issues and Streams

Both Issue and Streams are represented as basic classes, specific Issue inherit from the basic Issue class, specific Stream inherit from the Stream class. Associated with both classes, there is a factory class. The issue factory class is responsible for creating issues for the issue deserializing mechanism, the stream factory is responsible for creating streams according to some URLs.

3.2 Architecturally Significant Design Packages

The ERS system can be decomposed into a lightweight core and additional streams and issues.

- Additional streams typically link against more advanced transports, like MRS.
- Additional issues typically describe domain specific issues, for instance issues related to some hardware.

3.2.1 Core ERS

The core ERS package contains common issues and simple streams, typically streams that write issues to files or to standard template library streams. It has no dependencies to other software packages. It offers Issue classes to represent the following events

- Compile and runtime assertions, preconditions, unimplemented functions.
- Errors related to the ERS package

The core ERS system also offers simple issue streams to do serialize issues:

- Output to human readable text format.
- Output to tabulation separated text.
- Input and output to XML.

This package depends on the System package for input/output and related issues.

3.2.2 System package

The system package is a package that contains facilities to manipulate basic system level entities like:

- Files (existence, permissions, renaming, erasing)
- Executables (launching, parameter list manipulation).
- Environment (getting and setting environment variables).
- Memory mapped files (creation and manipulation).

All methods provided in the System package are ers-enabled, that is in case of errors, the method send back appropriate issues. In particular, the System package implements a whole hierarchy of POSIX issues. This package depends on ERS for core error handling facilities.

3.2.3 MRS ERS Stream

This package builds a facilities to stream ERS issues to the MRS service. It has dependencies on the MRS package and related services (IPC and CORBA), it also depends on the core ERS and the System package.

3.2.4 Thread Stream

This package offers a stream to be used between threads within an single process. This is based on a producer-consumer pattern. With worker threads producing issues and one or more error handling threads consuming them. Figure 2 illustrates this architecture.

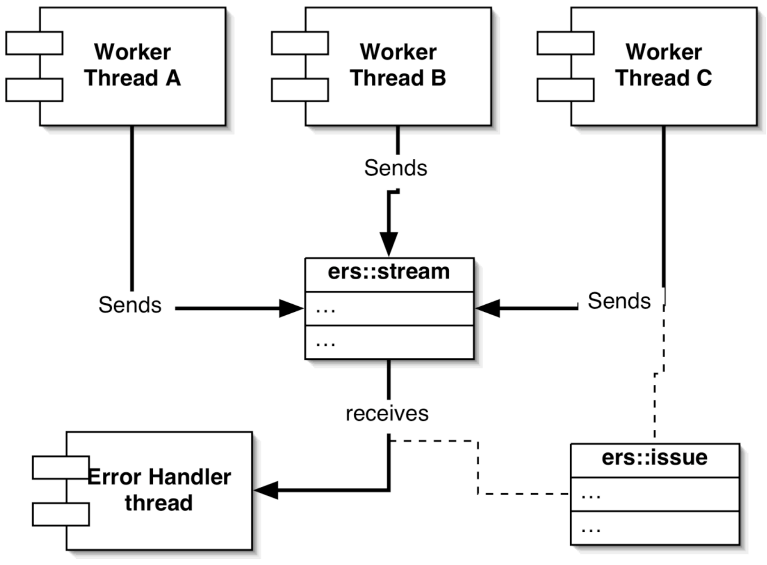

Figure 2: Thread stream

This package has dependencies on the common thread library.

4 Use-Case View

4.1 Use-Case Realizations

For simplicity's sake, the use-cases are presented in the C++, the same use cases are also relevant in the Java language.

4.1.1 Throwing an ERS issue

Application code encounters a problem and throws an ERS issue as an exception. The exception is constructed like a simple object, information about the context of the issue (time, code-location, etc) is inserted automatically into the object.

```
const int result = ::chmod(m_full_name.c_str(),permissions);
if (0==result) return ;
throw ERS_CHMOD_ERROR(m_full_name.c_str(),permissions) ;
```

4.1.2 Building a specific issue

Detector throws some specific issue to be handled by the error system. The constructor is customised to handle the specificity of the issue.

In order to build a custom issue, one needs to subclass the `ers::Issue` class. This subclass must define two constructor: one with no argument that is used by the factory method, and one that takes as parameters a Context and a severity level, plus whatever specific information that is needed. For each piece of data that needs to be stored, a string key must be defined. The class should also overload the `class_name` method to return a string key to identify the class.

```
#include "ers/Issue.h"
class ExampleIssue : public ers::Issue {
protected:
    ExampleIssue(const ers::Context& context,
                 ers::ers_severity severity) ;
public:
    static const char* CLASS_NAME ;
    static const char* PROCRASTINATION_LEVEL_KEY ;
    ExampleIssue();
    ExampleIssue(const ers::Context& context,
                 ers::ers_severity severity,
    int procrastination_level) ;
    virtual const char *get_class_name() const throw() ;
    int procrastination_level() const ;
} ; // ExempleIssue
#define EXAMPLE_ERROR(level) ExampleIssue(ERS_HERE,ers_error,level)
```

Each subclass of Issue should register a factory method so that deserialisation of issues is possible. Those methods should be declared in a anonymous namespace in order to avoid name proliferation

```
#include "ExampleIssue.h"
namespace {
    ers::Issue *create_issue() { return new ExampleIssue(); }
    bool registered =ers::IssueFactory::instance()\
       ->register_issue(ExampleIssue::CLASS_NAME,create_issue) ;
}
```

The class name should be as close as possible to the C++ class name to avoid confusions. One cannot rely on RTI information to build this string, as the specification allows it to be mangled or an empty string.

```
const char *ExampleIssue::CLASS_NAME = "ExampleIssue" ;
const char *ExampleIssue::PROCRASTINATION_LEVEL_KEY =
      "PROCRASTINATION_LEVEL" ;
```

The `get_class_name` method simply returns the key used to serialize the class. It should be the same as the one used to register the factory method.

```
const char *ExampleIssue::get_class_name() const throw() {
      return ExampleIssue::CLASS_NAME ;
} // get_class_name
```

The empty constructor is used by the factory method.

```
ExampleIssue::ExampleIssue() : ers::Issue() {}
```

A protected constructor taking as parameter context and severity should be provided for subclasses.

```
ExampleIssue::ExampleIssue(const ers::Context& context,
      ers::ers_severity severity) : ers::Issue(context,severity) {}
```

The user constructor takes custom parameters to set values that are specific to the issue. Those values are not stored in member variables. Instead, data should be stored inside the value table of the issue. This value table can be manipulated using provided methods.

At the end of the setup, the custom constructor should call method finish_setup with the textual description of the Issue, this method will finish the initialization of the Issue (some things cannot be done by the constructor) and setup the human readable message.

```
ExampleIssue::ExampleIssue(const ers::Context& context,
      ers::ers_severity severity, int procrastination_level) :
      ers::Issue(context,severity) {
      set_value(PROCRASTINATION_LEVEL_KEY,procrastination_level) ;
      finish_setup("Procrastinating");
} // ExampleIssue
```

Methods to get custom fields should simply call the appropriate values to set and get values in the value table.

```
int ExampleIssue::procrastination_level() const {
      return get_int_value(PROCRASTINATION_LEVEL_KEY);
} // procrastination_level
```

4.1.3 Catching an ERS issue and streaming it

An exception handler catches an ERS issues, changes it to a warning and streams it. No transformation is needed to the issue object.

```
try {
      …
} catch (Issue &e) {
      TabOutStream tab_out(path);
      tab_out << e ;
}
```

4.1.4 Encapsulating issues

An ERS issue can be defined as being the cause of another, and linked inside it. The link is preserved when Issues are serialised.

4.1.5 Catching a STL exception and encapsulating it

An STL exception is caught and is encapsulated in an ERS issue and thrown again. STL exeptions are not stored per-se in the Issue, only their description.

```
try {
      this->m_stream = new std::ifstream(filename) ;
      m_stream->exceptions(std::ios::failbit | std::ios::badbit);
      m_delete_stream = true ;
} catch (std::ios_base::failure &ex) {

      throw ERS_OPEN_READ_FAIL(filename,&ex);
} // catch
```

4.1.6 Assertions

An assertion statement throws an exception if the assertion is violated. The statement is a simple macro that depending on compiler options, either is compiled out, or sends an exception.

A condition is set on a given element of state or a parameter. For instance a given point needs to be valid.

A condition is set for a given internal value of the code. Here we know that if exec returns, then status variable should be negative. Assertion that take a string parameter support variable argument numbers and `printf` like formatting.

Different assertion statements are provided to check for preconditions, internal assertions, mark non-implemented code etc.

4.1.6.1 General Assertion

An assertion that does not fit in a specific category is checked.

```
const int status = execv(name,argv) ;
ERS_ASSERT(status<0,"exec returned status: %d (>=0)",status);
```

4.1.6.2 Precondition

A particular assertion that checks the parameter to a function.

```
System::MapFile::MapFile(const std::string &name, size_t size, size_t
offset, bool read_mode, bool write_mode, mode_t permissions) :
File(name) {
      ERS_PRECONDITION((read_mode || write_mode),
      "neither read nor write mode");
      ERS_PRECONDITION((size%page_size)==0,
      "Size %d is not a multiple of pagesize %d",
      (int) size, page_size);

      ERS_PRECONDITION((offset%page_size)==0,
      "Offset %d is not a multiple of pagesize %d",
      (int) offset,page_size);
```

4.1.6.3 Compile time assertions

The code checks some compile-time known value:

```
ERS_STATIC_ASSERT(sizeof(int)==4); // We check integers are 32 bits
```

If the condition cannot be verified at compile time, a compile error occurs:

```
error: aggregate `ers::Compile_time_error<false>
ERROR_ASSERTION_FAILED' has incomplete type and cannot be defined
```

4.1.6.4 Pointer checks

We check that a point returned from a library call is valid.

```
const char* p = strcat(a,b,sizeof(b));
ERS_CHECK_PTR(p); // We check the pointer is valid
```

4.1.6.5 Pointer precondition

A function checks that a pointer given as a parameter is valid

```
ers::File::File(const char* c_name) {
      ERS_PRE_CHECK_PTR(c_name); // We check the pointer is valid
      std::string name = std::string(c_name);
      set_name(name);
} // File
```

4.1.6.6 Non implemented function

Some code element is not implemented (yet).

```
 ERS_NOT_IMPLEMENTED();
```

4.1.7 Reading an Issue from a stream and throwing it

An issue is read from a stream and thrown as a exception. The read class has the correct C++ type and can be filtered using the C++ catch system.

```
Issue *i = xml_in.receive() ;
throw *i ;
```

4.1.8 Translating Framework specific errors

A framework notifies an error using its own error handling mechanisms, this error is translated and dispatched using ERS mechanisms.

```
const short x_severity = domError.getSeverity();
ers::ers_severity severity = ers_error ;
switch (x_severity) {
      case DOMError::DOM_SEVERITY_WARNING:
            severity = ers_warning ;
            break ;
      case DOMError::DOM_SEVERITY_ERROR:
            severity = ers_error ;
            break ;
      case DOMError::DOM_SEVERITY_FATAL_ERROR:
            severity = ers_error ;
            break ;
      default:
            NOT_IMPLEMENTED();
            break ;
      } // switch
std::string message = to_string(domError.getMessage());
ParseIssue issue(ERS_HERE,severity,message);
DOMLocator *location = domError.getLocation();
const int line_number = location->getLineNumber() ;
issue.offending_line_number(line_number);
std::string uri = to_string(location->getURI());
issue.file_name(uri);
DOMNode *node = location->getErrorNode();
std::string node_name = error_msg(node);
issue.offending_line(node_name);
return issue ;
```

4.1.9 Checking that a memory allocation is successful

Code checks that a memory allocation (via the `malloc` and `calloc` functions) did not return a null pointer (i.e failed).

```
char **argv = (char **) calloc(sizeof(char*),argclen) ;
ERS_ALLOC_CHECK(argv,sizeof(char*),argclen) ;
```

4.1.10 Checking that a system call succeeded

Most POSIX calls return a certain value, code checks that is value is correct.

```
const int status = ::setenv(c_key,c_value,1);
if (status<0) throw PosixIssue(ERS_HERE,ers::ers_error,"unable to set
environnement") ;
```

In many cases, some specific Issue are available for often used POSIX functions. The POSIX class handles the error information stored in the global variable `errno`.

```
const int status = ::close(m_map_fd);
if (status<0) throw System::CloseFail(ERS_HERE, m_full_name.c_str());
```

4.1.11 Debug Messages

User code logs some debug information. Log messages can be compiled out, discarded at runtime or sent to any issue stream.

```
if (node->getNodeType()==DOMNode::COMMENT_NODE) {
      std::string text = get_text(node) ;
      ERS_DEBUG_3(text.c_str()) ;
      return ;
} // Comment node
```

4.1.12 ERS aware wrappers

Wrappers provide ERS aware functions that offer low level functionality.

```
std::ostringstream stream ;
stream << "echo \""  << text << "\" > " << path ;
std::string result = System::Executable::system(stream.str());
```

The `System::Executable::system` method offers the same functionality than `int system(const char *string);` function, but throws Issue objects as exceptions in case of failure. Thus, users do not need to worry about error handling.

5 Process View

When used between processes, ERS streams are bound to the behaviour of the stream implementation. In general, inter-process stream implementations are built using event or message passing services, with typically one sender and one or more receivers. Issue routing can imply intermediate processes or hosts.

Figure 3 illustrates the architecture with and underlying MRS transport. The client process sends exceptions to a ERS stream, this stream is implemented using a MRS stream. This streams sends data to the MRS server. The server pushes the message to one or more MRS receivers. The client process hosts the ERS interface and the MRS client interface, the MRS is a different process, and the MRS receiver is a third process.

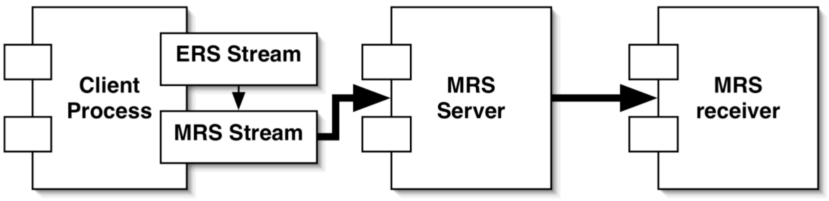

Figure 3: ERS Stream with MRS implementation

6 Implementation View

6.1 Overview

The implementation of the ERS is based around two core objects: issues and streams. The actual issue and stream functionality is build by subclassing those objects.

6.1.1 Issue

The issue class has to have the following features:

- It should be easy to subclass.
- It should be easy to serialize.
- It should be easy to transport using services like MRS.

In order to achieve those goals, issues are implemented as a table of key value pairs. Subclasses don't add new member fields to the object but insert new key-values into the hash-tables. This makes serialisation easy as a single mechanism is needed to serialize the hash-table (see 7.2). For simplicity, values are stored as strings, other values like Boolean or numbers are stored in textual format and parsed if needed.

The root Issue class is the default type of issue and can be used directly, although the usage of specialised sub-classes is recommended. The root class contains core functionality and method to get and set general properties of issues as defined in [1], among them:

- The severity of the issue (fatal, error, warning, etc…)
- The responsibility in case of chained issues
- The issue context (source code file and line, compiler, time).
- A textual, human readable error message that summarises the issue.
- Transience, i.e. if the error is transient or not.
- Completion, i.e. if the request that caused the error completed.

Those properties are represented as entries is the key-value table of the issue. See paragraph 4.1.2 for a discussion about sub-classing the Issue class for specific purposes.

6.1.1.1 Issue Factory

In order to deserialize Issues from a stream, we need a special class that knows what class to instantiate when given a textual representation of the class type. This class implements the Singleton and the factory pattern [2]. In order to work in C++ each Issue subclass must register a factory function with this central Issue factory.

6.1.1.2 Context

This class encapsulates the functionality representing the originating context of an issue. The context object contains the compile time information about where a exception is thrown, this includes the source code position (file, line), the encompassing function, compilation information (compiler, compile time), and general information (platform). Context objects are meant to isolate the Issue object from the macros that are used to collect context information. Context objects are built using specialised macros, and passed as an argument to Issues upon construction.

6.1.2 Stream

The root stream class is abstract and does not implement any behaviour, subclasses can implemented either the input stream functionality, the output stream functionality or both. The core library offers basic streams functionality to STL streams, more advanced streams are offered by different packages.

6.1.2.1 Stream Factory

The stream factory object implements the singleton and the factory pattern. The stream factory has two responsibilities:

- Manage the default streams for each severity level.
- Instantiate those streams according to URL defined in environment variables.

6.1.2.2 Human Stream

This stream is used internally by Issue in order to give a human readable textual representation. When such a representation is requested, the issue is streamed into this special stream, and the resulting text is returned as a string.

This textual representation is used when an Issue is sent to a STL stream using the << operator.

6.1.2.3 Tabular Separated Stream

This stream is used to give a more structured representation of issues. This can be used to potentially import issue in other programs that support the tab format. In order to avoid complexity in the streaming code, both key and values string should avoid the usage of the tab character.

6.1.2.4 XML Stream

The XML format is meant to be the default format for serialized issues. Streams to both read and write this format will be provided. This stream implies a dependency on an external XML library, the TDAQ standard is Xerces [3]. The format is described in Section 7.2.

6.2 Layers

6.2.1 Utility layer

This layer offers basic definitions for types, enumeration, function to read and parse data-types from strings. It also offers macros to handle compilation-time information to be put into issues, like compilation time, compiler, source-file, line number etc.

6.2.2 ERS layer

This layer offers the basic issue and stream object along with the subclasses described in the core ERS module (3.2.1)

7 Data View

Data can be stored in multiple formats, but the only format implemented by the core library are tabulation separated and XML.

7.1 Tabulation separated text

The tabulation format is meant as a simple human readable format for data.

```
KEY_NAME_1  "value text"
KEY_NAME_2  "value text"
```

The character between the key and the value is a tabulation character, each line is terminated by a carriage return character. This file format cannot handle issue chaining, in case of issue chaining, only the textual description of the cause exception is inserted with the following format:

```
CAUSE_TEXT  "basic_ios::clear(iostate) caused exception"
```

7.2 XML Format

The XML format is meant to be the main stable storage format and also be used when streaming using unstructured channels. The XML format is simple, as it only describes a list of key-value pairs. The proposed format is the following:

```
<issue>
<key>key name 1</key><string>value text</string>
<key>key name 2</key><string>value text</string>
…
</issue>
```

When issues are chained the value can be a <issue> tag instead of <string>, e.g:

```
<issue>
<key>key name 1.1</key><string>value text</string>
<key>key name 1.2</key><string>value text</string>
<key>ISSUE_CAUSE</key>
      <issue>
      <key>key name 2.1</key><string>value text</string>
      <key>key name 2.2</key><string>value text</string>
      …
      </issue>
</issue>
```

This XML format is very simple, here is the DTD for this format

```
<!ELEMENT issue (key,string,issue)>

<!ELEMENT key (#PCDATA)>

<!ELEMENT string (#PCDATA)>
```

8 Size and Performance

The ERS system per se will only run locally, so it does not have to handle issues like scaling. This does not mean that those issues can be ignored, simply that scaling is an issue for the transport and the aggregation system, not the error reporting system.

An important issue for the ERS library is size, as this code should in all parts of the system, code footprint is an primary concern. Thus the library has to include minimal functionality in its basic form.

In normal operations, ERS should impose minimal computing overheads. It should be possible to compile out as much as possible ERS code so that time critical code can control on what ERS facilities they rely on. On the other hand error handling code needs not to perform especially well, error handling should be the exception and thus optimising it makes no sense. Robustness and small footprint are more important for error handling code than performance.

In order to be usable in certain low level cases, it should also be possible to statically link the library into certain tools.

9 Quality

The goal of the ERS library is to improve the robustness of the general TDAQ software. Because of this the code has to be extremely robust and handle internal errors in the most graceful possible way. Also the ERS code should also serve as example of the usage of ERS error handling facilities, and thus should be clean and very well documented.