# The Error Reporting in the ATLAS TDAQ System

**Serguei Kolos**
University of California Irvine, USA
E-mail: serguei.kolos@cern.ch

**Andrei Kazarov**
CERN, Switzerland, on leave from Petersburg Nuclear Physics Institute, Kurchatov NPI, Gatchina, Russian Federation

**Lykourgos Papaevgeniou**
CERN, Switzerland

**Abstract.** The ATLAS Error Reporting provides a service that allows experts and shift crew to track and address errors relating to the data taking components and applications. This service, called the Error Reporting Service (ERS), gives to software applications the opportunity to collect and send comprehensive data about run-time errors, to a place where it can be intercepted in real-time by any other system component. Other ATLAS online control and monitoring tools use the ERS as one of their main inputs to address system problems in a timely manner and to improve the quality of acquired data.

The actual destination of the error messages depends solely on the run-time environment, in which the online applications are operating. When an application sends information to ERS, depending on the configuration, it may end up in a local file, a database, distributed middleware which can transport it to an expert system or display it to users. Thanks to the open framework design of ERS, new information destinations can be added at any moment without touching the reporting and receiving applications.

The ERS Application Program Interface (API) is provided in three programming languages used in the ATLAS online environment: C++, Java and Python. All APIs use exceptions for error reporting but each of them exploits advanced features of a given language to simplify the end-user program writing. For example, as C++ lacks language support for exceptions, a number of macros have been designed to generate hierarchies of C++ exception classes at compile time. Using this approach a software developer can write a single line of code to generate a boilerplate code for a fully qualified C++ exception class declaration with arbitrary number of parameters and multiple constructors, which encapsulates all relevant static information about the given type of issues. When a corresponding error occurs at run time, the program just need to create an instance of that class passing relevant values to one of the available class constructors and send this instance to ERS.

This paper presents the original design solutions exploited for the ERS implementation and describes how it was used during the first ATLAS run period. The cross-system error reporting standardization introduced by ERS was one of the key points for the successful implementation of automated mechanisms for online error recovery.

## 1. Introduction

ATLAS [1] is one of the four major experiments at the Large Hadron Collider accelerator at CERN. The ATLAS Trigger and Data Acquisition (TDAQ) [2] system selects and transports physics data from the 1600 detector read-out links to the mass storage. The system is composed of about 20K applications distributed over 3K computers. Controlling and monitoring such system requires flexible and reliable services for error reporting and handling both inside individual applications and between applications. The Error Reporting Service (ERS) gives software applications the opportunity of collecting and reporting comprehensive data about run-time issues as well as subscribing to the messages produced by the other applications. The ATLAS online control and monitoring tools use ERS as one of their inputs for getting information about the TDAQ system problems and reacting to them in a timely manner for improving the quality of acquired data. Figure 1 shows the main flow of the ERS messages in the TDAQ system.

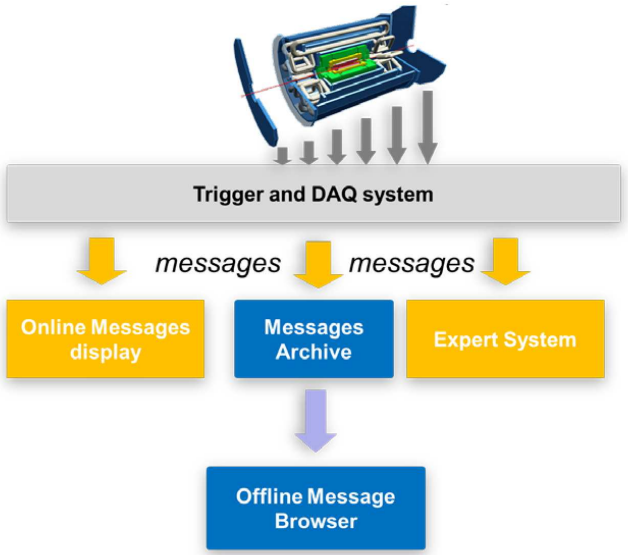


**Figure 1.** The flow of the ERS messages in the TDAQ system.

ERS is available in all programming languages used by the ATLAS software: C++, Java and Python. While the system design is the same for all of them, the implementations have their peculiarities which are defined by the specific features of the given programming language as discussed in the following sections.

## 2. The Error Reporting Interface

ERS interface is build around three main concepts: the Output Stream, the Issue and the Input Stream. An Issue contains description of a specific problem which can be reported to the Output Stream and can be retrieved, in the context of another application, via the Input Stream.

### *2.1. The Output Streams*

The Output Stream is a simple interface with a single function for sending arbitrary messages to that stream. There are six types of streams corresponding to different levels of severity of the reported issues: `DEBUG`, `LOG`, `INFO`, `WARNING`, `ERROR` and `FATAL`. The severity of any issue is established by the type of stream to which it has been sent. While ERS provides several default stream implementations, which can be used out of the box, it is an open framework which allows plugging in new implementations at any moment. Thus, new message destinations can be added without touching the frameworks code. Each Output Stream interface implementation is responsible for:

- taking an action which is specific to that stream. This can be sending the given issue to an appropriate output device, e.g. standard output, database or a mobile phone,
- deciding whether the given issue has to be propagated further through the chain of stream implementations or immediately suppressed. This feature allows implementing the issue filtering.

*2.2. The Output Streams Configuration*

The destination of the ERS messages depends solely on the run-time environment, in which the online applications are operating. Depending on the actual configuration, the messages which applications send to ERS may end up in a local file, a database, a message passing middle-ware, or in any other output which is supported by ERS. New output devices can be added to ERS as plugins without touching the ERS code. Stream configuration is very flexible and can be defined for each particular stream type at the level of an individual software process by setting the appropriate environment variables. A value of such variable is a comma-separated list of tokens where each token is a well defined ID of a stream implementation class optionally followed by the list of initialization parameters for that implementation. For example the Error stream can be configured via application environment in the following way:

```
TDAQ_ERS_ERROR="stderr, filter(Tile), abort"
```

In this case all errors will be sent to the standard error stream and any issue originated from the Tile Calorimeter subsystem will immediately abort the running application.

*2.3. The Issue Class*

The main class for reporting problems is called Issue. This class inherits from the std::exception in order to provide compatibility with the standard C++ library. Any custom ERS exception can be caught as std::exception.

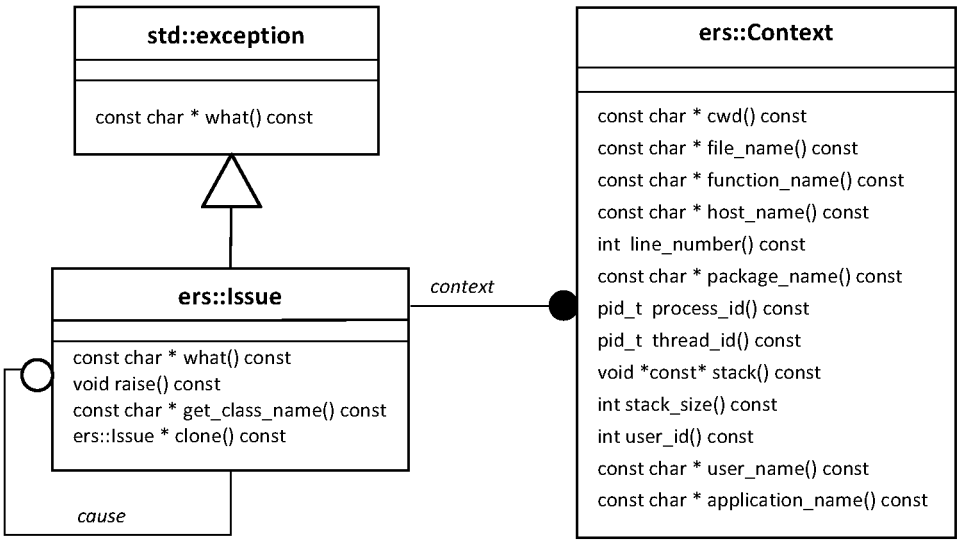


**Figure 2.** The main ERS public API classes.

The Issue class is abstract. Any software package which has to report an error must declare a specific issue class by inheriting it from the abstract ers::Issue or alternatively reuse an

existing issue class from another package. In this case each kind of problem is described by the corresponding C++ class, which facilitates the application of the expert system techniques for errors analysis and decisions taking procedures. As shown in Figure 2, each issue has the Context attribute which describes the place in the code where the issue has occurred. In C++ this information is provided by the special macro `ERS_HERE`, which must be the first argument to each Issue constructor. In Java and Python such information is extracted in the Issue constructor itself, so no additional arguments is used. An issue may contain a pointer to another issue which provides more detailed information on the cause of the problem. Such chain of issues may have arbitrary depth thus giving any required level of details for the problem description.

*2.4. The Input Stream*

Within the same process ERS issues are passed using the programming language exception mechanism, e.g. `try...catch` in C++ and Java and `try...except` in Python. For the inter-process issue handling ERS defines the asynchronous `Input` stream interface for receiving ERS Issues reported by other applications. The idea is that for each `Output` stream implementation one can implement a corresponding `Input` stream so that it will be capable of receiving the issues reported to its `Output` counterpart. For a pair of such streams a user can simply register a callback function with the `Input` stream implementation to get all issues reported by all processes to the corresponding `Output` stream.

## 3. C++ ERS API

Having well defined and strictly typed errors is highly desirable for simplifying the system maintainability and absolutely indispensable for replacing human operator with expert system. On the other hand writing issues classes declarations would have been tedious and error prone. To overcome this problem ERS uses the mind breaking BOOST Preprocessor [3] package. Despite conventional opinion the usage of macro constructs in C++ may be extremely useful and convenient, drastically reducing the amount of code which has to be written and improving the code quality and maintainability. The following two macros have been designed for the C++ ERS API and can be used for declaring component specific exception classes.

```
ERS_DECLARE_ISSUE(namespace, class_name, message, class_attributes)

ERS_DECLARE_ISSUE_BASE(namespace, class_name, base_class_name,
                       message, class_attributes, base_class_attributes)
```

The first macro can be used for declaring classes inheriting the ers::Issue class, while the other one can declare classes derived from the other macro-defined ERS exceptions. The most interesting part in these macros is the ability of passing arbitrary number of the class attributes via the `class_attributes` and `base_class_attributes` parameters. This is where the BOOST Preprocessor package, which implements advanced parsing and looping functions for standard C macro, has been exploited. A class attributes is the sequence of the type and name macro tuples which looks like:

```
((attribute1-type) attribute1-name)
((attribute2-type) attribute2-name)
...
((attributeN-type) attributeN-name)
```

The macro generate C++ classes with the constructors containing a given number of parameters. A C++ developer will need to provide appropriate values for all the attributes when constructing the corresponding exception. Those values will be stored in the exception and inserted into the

main message of the issue, which is declared only once at the issue definition time. This C++ class also provides accessor functions, like `get_attributeX-name()` for all declared attributes. For example placing the following declaration into a C++ file:

```
ERS_DECLARE_ISSUE(
  io,
  FileIssue,
  "Basic issue with '" << file_name << "' file",
  ((const char *)file_name ))
```

will produce the C++ code like:

```
namespace io {
  class FileIssue {
    FileIssue(const ers::Context & context , const char * file_name );
    FileIssue(const ers::Context & context , const char * file_name,
              const std::exception & cause );

    const char * get_file_name() const;
  };
}
```

The `ERS_DECLARE_ISSUE_BASE` macro is similar to the `ERS_DECLARE_ISSUE` one but allows declaring hierarchies of exceptions classes. For example one can declare the `io::PermissionDenied` and `io::CantOpenFile` exceptions which inherit the `io::FileIssue`in the following way:

```
ERS_DECLARE_ISSUE_BASE(
  io,
  PermissionDenied,
  FileIssue,
  "Insufficient privileges for '" << operation << " '" << file_name
          << " file which has '" << permissions << "' permissions",
  ((const char *)file_name ),
  ((int)permissions) ((const char*)operation))

ERS_DECLARE_ISSUE_BASE(
  io,
  CantOpenFile,
  FileIssue,
  "Cant open file " '" << file_name << "' file",
  ((const char *)file_name ),
  ERS_EMPTY )
```

*3.1. Catching and Reporting Issues*

There are six functions for reporting issues to different ERS streams, e.g. ers::debug(), ers::log(), ers::info(), ers::warning(), ers::error() and ers::fatal(). The last three of them accept only strictly typed messages, i.e. objects of a type inherited from the ers::Issue. In this way the important messages are imposed to be strictly typed, allowing the TDAQ controlling and monitoring systems to rely on their content and meaning. An example of the usage of the three functions for reporting issues which accept only strictly typed messages is shown below.

```
  try {
    open_file( file_name );
  }
  catch ( io::PermissionDenied & ex ) {
    ers::warning(io::CantOpenFile( ERS_HERE, file_name, ex ));
  }
  catch ( io::FileIssue & ex ) {
    ers::error( ex );
  }
  catch ( std::exception & ex ) {
    ers::fatal(io::FileIssue( ERS_HERE, file_name, ex ));
  }
```

The other three streams accept both the ers::Issue instances and arbitrary dynamically constructed information. For the latter ERS provides three macros: `ERS_DEBUG`, `ERS_LOG`, `ERS_INFO`, which can be used to pass arbitrary information to ERS without defining a new issue type. For example:

```
  ERS_DEBUG( 1, "test debug macro " << 12345 );

  ERS_LOG( "So far " << event_number << " events have been collected");

  ERS_INFO( "The run " << run_number << " has been started" );
```

## 4. Java ERS API

Java ERS API supports exceptions which are derived from the `ers.Issue` as well as the standard Java exceptions, which can be reported by third party libraries, e.g. derived from the `java.lang.Exception`.Contrary to C++ Java does not have macro preprocessor so exception classes have to be declared manually, e.g.:

```
public class ConnectionIssue extends ers.Issue {
  public String reason ;
  public int port_number ;

  ConnectionIssue (String reason, int port_number) {
    super("Connection to %p port failed because of %s", port_number, reason);
  }

  ConnectionIssue (String reason, int port_number, ers.Issue cause) {
    super("Connection to %p port failed because of %s", port_number, reason, cause);
  }
};
```

Note that one has to declare two constructors: one takes another exception as its last argument to allow chains of the issues and the other one has only the actual issue specific parameters to be used for creating the first level exception.

### *4.1. Reporting Issues*

Reporting of ERS Issues in Java is done using static functions declared in the `ers.Logger` class. The *Debug*, *Info* and *Log* functions support plain strings as their arguments, other methods only accept `ers.Issue` or `java.lang.Exception` objects. For example:

```
...
catch { java.net.SocketException ex } {
  ers.Logger.error(new ConnectionIssue("Socket is not available", port, ex)));
}
catch { java.lang.StackOverflowError ex } {
  ers.Logger.fatal(ex) ;
}

ers.Logger.debug(2, "Verbose debug message");
ers.Logger.log("A message to be logged in log file");
```

*4.2. Log4J Mapping to ERS*

Most of third-party java libraries widely used in TDAQ software use de-facto standard Java logging framework `log4j`(http://logging.apache.org/log4j/2.x/) for reporting messages. To allow log4j messages to be routed over ERS, Java ERS library includes an implementation of log4j *appender*, which converts log4j messages to ERS issues and redirects them to the appropriate ERS streams. For example the following `log4j` configurations routes all `DEBUG` messages to ERS stream.

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
   <appender name="ERS" class="ers.log4j.ERSAppender">
    <layout class="org.apache.log4j.SimpleLayout"/>
   </appender>

   <root>
    <level value="DEBUG" />
    <appender-ref ref="ERS"/>
   </root>
</log4j:configuration>
```

## 5. Handling Issues in the Distributed Environment

In order to report and receive ERS issues in the distributed TDAQ system, two special `Input` and `Output` streams have been implemented. They are based on the Message Transfer System (MTS) [4], which is the CORBA [5] based distributed message passing middle-ware developed in the ATLAS TDAQ project.

## 6. Conclusion

During the Run 1 ERS has proved to be extremely useful for both error detection and problems analysis. In total about 30M ERS messages have been produced by the TDAQ system processes and reported via ERS. All those messages have been archived to the Oracle [6] database. A dedicated application called Log Browser has been provided for browsing the content of that database. At run time about 10 different types of ERS issues have been used by the ATLAS online expert system for error detection and handling during data taking.

For the upcoming data taking period many more ERS messages are going to be used by the online expert system to improve the automation of the data taking. In addition to that the archived system back-end will be changed to Splunk [7], which is an easy-to-use web interface and powerful enterprise platform for analyzing machine generated data, like log files, message archives, etc. This allows having access to the archived messages from all over the world without the need of installing any ATLAS specific software.

## References


[1] ATLAS Collaboration, The ATLAS experiment at the CERN large hadron collider, Journal of Instrumentation, vol. 3, no. 08, p. S08003, Aug. 2008. Available: http://iopscience.iop.org/1748-0221/3/08/S08003

[2] ATLAS Collaboration, ATLAS high-level trigger, data-acquisition and controls, CERN, Technical Design Report ATLAS-TDR-016 CERN-LHCC-2003-022, 2003. Available: http://cdsweb.cern.ch/record/616089/

[3] Boost Preprocessor library home page http://www.boost.org/doc/libs/release/libs/preprocessor/

[4] A. Kazarov et al., A Scalable and Reliable Message Transport Service for the ATLAS Trigger and Data Acquisition System, *Proc. of the 19th Real-Time IEEE Conf., Nara, Japan, May 2014*. Available: http://cds.cern.ch/record/1703434

[5] CORBA home page http://www.corba.org

[6] Oracle home page http://www.oracle.com

[7] Splunk home page http://www.splunk.com