The ALICE DAQ infoLogger

S. Chapeland1, F. Carena1, W. Carena1, V. Chibante Barroso1, F. Costa1, E. Dénes2, R. Divià1, U. Fuchs1, A. Grigore1,3, C. Ionita1, C. Delort1, G. Simonetti1,4, C. Soós1, A. Telesca1, P. Vande Vyvre1, and B. Von Haller1 for the ALICE collaboration

1 CERN, European Organization for Nuclear Research, Switzerland
2 Institute for Particle and Nuclear Physics, Wigner Research Center, Budapest, Hungary
3 Polytechnic University of Bucharest, Romania
4 Dipartimento Interateneo di Fisica ’M. Merlin’, Bari, Italy

E-mail: Sylvain.Chapeland@cern.ch

Abstract. ALICE (A Large Ion Collider Experiment) is a heavy-ion experiment studying the physics of strongly interacting matter and the quark-gluon plasma at the CERN LHC (Large Hadron Collider). The ALICE DAQ (Data Acquisition System) is based on a large farm of commodity hardware consisting of more than 600 devices (Linux PCs, storage, network switches). The DAQ reads the data transferred from the detectors through 500 dedicated optical links at an aggregated and sustained rate of up to 10 Gigabytes per second and stores at up to 2.5 Gigabytes per second. The infoLogger is the log system which collects centrally the messages issued by the thousands of processes running on the DAQ machines. It allows to report errors on the fly, and to keep a trace of runtime execution for later investigation. More than 500000 messages are stored every day in a MySQL database, in a structured table keeping track for each message of 16 indexing fields (e.g. time, host, user, ...). The total amount of logs for 2012 exceeds 75GB of data and 150 million rows. We present in this paper the architecture and implementation of this distributed logging system, consisting of a client programming API, local data collector processes, a central server, and interactive human interfaces. We review the operational experience during the 2012 run, in particular the actions taken to ensure shifters receive manageable and relevant content from the main log stream. Finally, we present the performance of this log system, and future evolutions.

1. Introduction

1.1. The ALICE experiment

ALICE (A Large Ion Collider Experiment) [1][2], is the heavy-ion experiment designed to study the physics of strongly interacting matter and the quark-gluon plasma at the CERN LHC (Large Hadron Collider) [3]. It primarily targets heavy-ion lead-lead collisions (Pb-Pb), but it also has a substantial physics program with proton-proton (pp) and proton-ion (pA) collisions. The experiment has been designed to cope with the highest particle multiplicities expected for Pb-Pb reactions. The spectrometer includes high resolution tracking (silicon detectors, large time-projection chamber), particle identification, and triggering elements. It features two large magnets, a main solenoid and a dipole on the Muon arm. ALICE consists of 18 detectors, being able to take data independently
(standalone operation) or in global partitions (set of detectors running together). The operation and
readout of the detector is handled by the ALICE online systems: Data-Acquisition (DAQ), trigger,
High-Level Trigger (HLT), and Detector Control System (DCS). They are overseen and synchronized
by the Experiment Control System (ECS).

1.2. The ALICE Data-Acquisition system

The ALICE DAQ [4][5] handles the data flow from the sub-detector electronics to the archiving on
taxe. A first layer of computers, the Local Data Concentrators (LDCs), reads out the event fragments
from the optical Detector Data Links (DDLs). Several DDLs can be connected to the same LDC, and
several LDCs may be needed to collect the data from a single sub-detector. The event fragments
aggregated in sub-events are then transferred to a second layer of computers, the Global Data
Collectors (GDCs), in charge of performing the event building. The same GDC receives all the
fragments of a given event, and assembles them in a full event, which is then recorded to a transient
storage before being migrated to tape. Each uninterrupted data taking period is called a run, ranging
from few minutes to many hours, with the same hardware and software configuration. There can be
several runs in parallel, with different detectors running together or standalone. Since 2010, the
ALICE DAQ recorded 6.5 PB of physics data.

1.3. Runtime environment

The DATE software [6] is a set of distributed processes running on the DAQ nodes to handle the data
flow described above, from the detector links to the permanent storage. In addition, DATE includes
the central services ensuring the configuration and control of the numerous runtime components. The
DAQ farm also hosts a number of services not directly part of the data flow but necessary to the
experiment online facilities: the ECS, the ALICE Configuration Tool (ACT), the experiment logbook,
the Data Quality Monitoring (DQM), and the operator consoles in the ALICE control room. During
the last operations before the LHC 2013 shutdown, the DAQ computing hardware consisted of
roughly 150 LDCs, 80 GDCs, 500 input links, 80 disk arrays, 60 desktop PCs, and 30 servers with
various purposes. The other online systems (DCS, CTP, and HLT) run on dedicated machines
managed by other teams.

During the last heavy-ion data taking period, from 20th January to 10th February 2013, over 6000
permanent and transient DAQ processes were active in average during a run. In this rich and
heterogeneous environment, it is challenging but necessary to have an easy access to the information
reported by such a large number of processes.

2. The DAQ logging system

The infoLogger package provides facilities to generate, transport, collect, store and consult log
messages from all distributed software components running on the ALICE DAQ system. It provides
an interface to inject logs, a central repository to store the messages, and user interfaces to display and
query them.

The current implementation was initiated in 2004 as an upgrade of the existing infoLogger package
in the previous DATE version. It aimed at fulfilling the ALICE DAQ processes monitoring and
reporting needs in a scalable tool. Although some other logging solutions were considered, none was
matching all our requirements (among others: central collection, multithreaded server, database
backend, web access, local buffering, custom log fields) or mature enough at that time, hence the need
for our in-house development.

2.1. Architecture

Figure 1 shows the overall architecture of the infoLogger system. A process calling a function of
the infoLogger library sends the message to the local infoLoggerReader daemon. This process collects
all the messages of the node where it runs, and sends them to a central infoLoggerServer daemon,
which stores the received messages in a database. The infoBrowser user interface allows reading messages, either stored in the database or received on-line by the central server.

2.2. Log message structure
Each log message handled by the infoLogger system is structured by an extensive set of attributes. Some of them are provided by the client creating the log message; others are set automatically by the infoLogger API when it is called. Each log message consists of the following fields:

- **Severity**: the information level of each message. This can be one of Information (messages concerning normal running conditions), Warning (to report a condition which could be the source of problems), Error (when an abnormal situation has been encountered), Fatal (when an unrecoverable situation has been detected, usually causing the end of the current run).
- **Level**: an integer used to prioritize messages, essentially based on their target audience. Typically, it allows to tag messages (from higher level to lower level) for operators, DAQ support team, DAQ developer, and debugging information. A range has been defined for each of these categories, so that a finer granularity is possible for further ordering within a category.
- **Timestamp**: time of the message creation, with microsecond resolution, as provided by the local operating system where the message is created.
- **Host name**: host where the message was created (physical name of the machine).
- **Role name**: DAQ role where the message was created (logical name of the DAQ entity running the process, e.g. the name of a LDC or GDC).
- **Process ID**: operating system identifier of the process creating the message.
- **User name**: user running the process creating the message.
- **System**: system originating the message. For all DATE processes, this is set to DAQ. As the infoLogger facilities are shared with other systems, it can be also set to ECS, TRG, HLT.
- **Facility**: the activity family, usually the DATE package name creating the message. This can be for example readout, recorder, runControl, or operator in case of a message coming from the command line.
- **Detector**: the name of the detector associated to the running process.
- **Partition**: the name of the global partition associated to the running process.
- **Run number**: the run number associated to the message, if any.
- **Error code**: an error code to tag the message. This is used to point to appropriate documentation and procedures for the operator. This also allows making statistics on error occurrences, even if the message content is different (for example when variables are printed in the message).
- **Source line**: line number where the message is issued in the source file.
- **Source file**: the name of the source file issuing the message. It allows (with previous field) to easily trace which part of the code issues the message.
- **Message**: the log information. It is a text string.

The client process calling the API typically provides the severity, level, error code, and message. All other fields are usually inferred automatically from the compiling and runtime environments. Although some of the fields are associated with specific ALICE DAQ semantics, the infoLogger is a general purpose logging system and can be used in other environments (with or without modifying the list of fields, the simplest being to leave specific fields unused). It has for example been adopted for the ALICE HLT as well. The infoLogger is already available as a standalone package, and it is also planned to fully decouple the infoLogger from DATE bindings in order to ease its dissemination (with appropriate documentation on how to change the fields by editing only a few well-identified source code sections).

### 2.3. Implementation

The infoLogger is written in C, and the native client API is provided in C. Extensions for other languages (e.g. Tcl/Tk) are generated using SWIG [7]. The client API provides a set of functions to send messages. The API remained backward compatible since its initial release in 2004, but was augmented together with the new features added. All message injection functions are eventually used for the same purpose, but differ in the calling arguments depending on the fields being set by the client. Some functions allow to set once for all some of the fields for a running process, in order to simplify further infoLogger calls. Finally, a few control functions allow to define infoLogger parameters for the process, like timeouts and message filters settings (e.g. to drop messages if their level detail is too high).

The API has a built-in protection to detect messages flood, when a client sends too many messages per second (which can for example occur if there is a bug in a loop). Corresponding thresholds are adjustable, and default values are set to maximum 500 messages in a second, or 1000 messages in a minute. Logs are afterwards redirected to a local file (with a maximum size, after which they are dropped). Normal operation resumes if the flow goes back to acceptable levels.

The API provides the possibility to log to a file (instead of the standard server) for verbose processes which logs do not need to be centrally accessible. It includes auto log rotation based on file size or number of messages, with different file formats (showing different subsets of infoLogger fields).

The client package includes a command line tool used to pipe to infoLogger the output of processes (mainly shell scripts) which are not using the infoLogger API.

The infoLoggerReader is started at boot time on all machines. If the infoLoggerReader process is not reachable by a client process (i.e. the initial issuer of the log message), the client process tries to
start a new instance of the infoLoggerReader. If this fails, messages are written to a local file on disk to avoid loosing them. Connection between local client and reader is done through a named pipe.

Messages collected by infoLoggerReader processes are stored in a persistent disk FIFO, which ensures their (delayed) delivery to the central infoLoggerServer in case it is unavailable at a given point in time. The infoLoggerReader ships data to the central server by a TCP/IP link with custom protocol. The list of fields has increased with time and needs. Protocol has been adapted to be easily extended, and includes versioning to allow decoding of different versions by the same server (e.g. when some clients still run an older version of the infoLogger library).

The infoLoggerServer runs a thread waiting for new infoLoggerReader connections and reading out the existing sockets. Messages collected are then inserted in a common message FIFO, for insertion in a relational database table by one or several (number can be configured) independent threads. It also forwards the incoming messages to a dispatch thread, which duplicates them to online subscribers (as described later).

The database is implemented with MySQL (selected in 2004 for its insert performance and ease of deployment), and the table structure follows the message structure (i.e. one column per message field). Insertion is done with MySQL C API and prepared statements. All fields are indexed to optimize query performance, although it considerably increases the database size on disk. The infoLogger message table is partitioned by hash on the timestamp column (with a division factor, so that there is one partition per day or so, i.e. keep less than 1 million messages per partition) in order to speed-up query performance: most queries being based on time, MySQL effectively implements partition pruning and does not scan partitions outside the time range. Otherwise, query performance may degrade as table size grows over few millions of rows. The messages table is manually archived every few months to keep it at a reasonable size. The table is simply renamed, and a fresh blank messages table created. Archived tables are still accessible for queries.

Because of the buffers and many-to-one links in the chain, the order of message insertion is not guaranteed; only the message timestamp (set at earliest point in the chain by the client API) is reliable to order messages coming from a given machine. The accuracy of clock synchronization is critical when correlating events from different nodes, and is ensured by a classical NTP setup for all the DAQ machines.

The display of the messages is tackled by the infoBrowser Graphical Human Interface, shown in figure 2. It works both in offline and online mode. In offline mode, it queries the messages available in the database. In online mode, the infoBrowser connects by TCP/IP directly to the infoLoggerServer, which dispatches a copy of all incoming messages to the subscribers. In both modes, the user can define filters based on the message fields’ content in order to select or exclude what messages should be displayed. In online mode, the filtering is done in the browser, i.e. the server sends the full log data stream. The infoBrowser is available as a Tcl/Tk application used in the runtime environment, and as a web interface (PHP/HTML) to ease remote (and secure) access to the ALICE DAQ logs. The web interface only allows offline queries to the database.

The infoBrowser has a direct connection with the DAQ Wiki for the resolution of problems: when an error is reported, the corresponding documentation is reachable from a single click in the user interface. The assignment of error codes is maintained with the infoLogger source code in CVS. Each DAQ package is assigned an error code range, and documents new error codes in a central file (together with the procedures in the Wiki, in structured pages providing for each error code: component, description, urgency, operator and expert actions to be taken, support procedures to be followed, who should be contacted, link to extra documentation, etc).
2.4. Operation

In production, the infoLoggerServer receives logs from 350 infoLoggerReader processes, and dispatches them to around 40 online infoBrowsers. During the 2012-2013 data taking period, it collected on average 400000 messages per day (800000 per day during most active periods like the heavy ion run, with daily peaks up to 1.6 million messages in the same day). In total, more than 160 millions of messages have been recorded, as seen in figure 3. This amounts to 83GB of MySQL data files, including indexes which account for 53 percent of the log database size.

Given the huge number of log messages received (typically between 10000 and 50000 per run), a review was performed in order to reduce the number of messages, in particular those visible for the operators. These recommendations were implemented in 2012, focusing on the improved tagging of messages in all DAQ components. This ensured that the DAQ operator now receives an acceptable number of messages (about a hundred in a normal run) by proper filtering. Since the developers preferred to keep a large amount of debugging messages, and since the log system could cope with it, we decided not to pursue efforts on message reduction for the time being. Anyway, these messages are by default filtered out for standard operations, but still accessible in case they are needed for investigation. As a comment, when opening a central log system, it should be taken into consideration that it may be difficult to control what eventually gets into it.

The infoLogger is the main entry point for processes to report runtime errors, and the most visible to the experiment operator. However, problems which are likely to sustain for a while or requiring
some non-trivial intervention, are forwarded and then followed-up in Orthos [8], the ALICE DAQ alarm system.

As regards performance, the log API overhead has been measured around 4 microseconds to inject a log message from client to reader, on a standard desktop machine. The latency for a single message log issued from the command line to display in the infoBrowser is around 1s, and is the same for a burst of 1000 message (all 1000 messages are visible in the infoBrowser within the same second).

Insertion speed in the database reaches 6000 messages per second, with a partitioned InnoDB table and all indexes. For comparison, the insertion speed is 9000 messages per second without indexes, and reaches 12000 messages per second with MyISAM storage engine and no index.

Good query performance is ensured by the multiple indexes, at the cost of an affordable increased disk space (and slight insertion time overhead). Taking a random day (by defining timestamp upper and lower limits in the SQL ‘where’ clause) with around 80000 messages, the following performance is observed (for un-cached queries): scanning (e.g. count all rows) the full logs takes less than 0.5 seconds; more complex SQL operations, like grouping by facility or host, take less than 2 seconds; retrieving the full data (about 200MB) is done in 8 seconds (or a read speed of 100000 rows per second). Complex queries to extract yearly statistics on the full log data set take more time, but results are available within minutes.

This large amount of logs generated locally is also very convenient to spot issues at a global level. It is for example possible to detect some abnormal situations by checking the amount of log messages produced by each component, and to measure unexpected variations by comparison with the other components of the same type. Large log producers are indeed often related to a malfunction. Another useful analysis is to monitor the recurrent error codes in order to identify software bugs or hardware failures. Currently done manually on punctual occasions, we aim at automatizing the data mining opportunities made available by this central logging system. The first step is to generate real-time and daily reports on a web page, and possibly to directly raise alarms in the Orthos system based on the message correlations observed. A prototype of a web-based PHP/HTML plotting and reporting tool is already working.

3. Conclusion

We presented the infoLogger, the software package used to generate, transport, collect, store and consult log messages from all distributed software components in the ALICE DAQ and other online systems. The infoLogger facilities have scaled according to needs since 2004, and the framework has demonstrated to be flexible enough to easily accommodate new requirements (e.g. more log fields). Future developments will be done to provide automatized data mining tools in order to spot suspect logging behaviors at a global level and feed the alarm system accordingly.

References
[1] ALICE Collaboration 1995 Technical Proposal, CERN-LHCC-1995-71
[2] ALICE Collaboration 2008 The ALICE experiment at the CERN LHC, JINST 3 S08002
[3] Evans L and Bryant P 2008 LHC Machine, JINST 3 S08001
[4] ALICE Collaboration 2004 The Technical Design Report of the Trigger, Data-Acquisition, High Level Trigger, and Control System, CERN-LHCC-2003-062
[5] Altini V, Carena F, Carena W, Chapeland S, Chibante Barroso V, Costa F, Divià R, Fuchs U, Makhlyueva I, Roukoutakis F, Sossmaier K, Soós C, Vande Vyvre P and von Haller B 2009 Commissioning the ALICE Experiment, Proc. CHEP 2009 (Prague, CZ: JPCS)
[6] Schossmaier K et al., The ALICE Data-Acquisition Software Framework DATE V5 , Proc. CHEP 2006 (Mumbai, IN)
[7] Simplified Wrapper and Interface Generator http://www.swig.org
[8] Chapeland S et al., 2012 Orthos, an alarm system for the ALICE DAQ operations, Proc. CHEP 2012 (New York, USA: JPCS)