GENERIC LOGGING LAYER FOR THE DISTRIBUTING COMPUTING

V. Fine*, J.Lauret, G..Van Buren
BNL, Upton, NY 11733, USA

Abstract

Most HENP experiment software includes a logging or tracing API allowing the display in a particular format of important feedback coming from the core application. However, inserting log statements into the code as a low-tech method for tracing the program execution flow often leads to a flood of messages in which the relevant ones are occluded. In a distributed computing environment, accessing the information via a log-file is no longer practical as the approach fails to provide runtime tracing.

Running a job involves a chain of events where many components may be written in diverse computing languages which do not offer a consistent and easily adaptable interface for logging important events.

We describe an approach based on a new generic layer built on top of a logger family derived from the Jakarta log4j project that includes the log4cxx, log4c, and log4perl packages. This provides consistency across layers, packages, platforms, and frameworks.

LOGGING API FOR HENP APPLICATION

Most HENP experiment software includes a logging or tracing API allowing the display in a particular format of important feedback coming from the core application.

Typical so-called “HENP production” jobs initiate a chain of events where many components involved are often written in diverse languages which do not offer a consistent and easily adaptable interface for logging important events.

Practical GRID case example

Most of the experiment frameworks offer a special (sometimes even rather sophisticated) layer whose aim is to provide the log messages from within the code which is executed, allowing tracing of the program execution flow. This output is typically redirected to a log file. In a distributed computing environment, this standard approach of accessing the information via a log file is insufficient as it fails to provide runtime tracing.

In fact, the problem is more complicated than it appears, as a typical “grid job” is often a complex workflow of meta-jobs composed of several sub-jobs running in parallel. Each given sub-job may be redirected to a different computing resource and one may want to have the ability for an individual job to trigger some action (abort the entire task or meta-job, invoke a recovery service, alter workflow, etc.). This action may be based on the program output or log itself, providing the entire workflow can be traced and individual jobs’ output related to one another.

To accommodate for the desirable components, we first need to provide the necessary tools to collect the information. Let us call such a tool a “logger”.

Consistency requirements

A meta-job requires a “meta-log” (Figure 1) to allow detailed inspection of any or all steps of the process. But a global overview of the process may require an abridged summary, necessitating the capability to disable certain log statements while allowing others to print unhindered. This assumes that the logging space, that is, the space of all possible logging statements, is categorized according to the framework developer-chosen criteria. In addition, one would like in a summary not to display or consider repetitive, which entails filtering facilities. Finally, an event severity and/or verbosity should accompany the display of every event (or message).

Jakarta log4j project

Searching for a suitable logger component for the STAR Grid production environment we found that an approach based on a new generic layer, built on top of a logger family derived from the “Jakarta log4j” [1] project, may provide consistency across packages and frameworks. “Jakarta log4j” is a community-supported project that includes the log4cxx, log4c, and log4perl packages (thereby supporting multiple computing languages) and encompassing four main component types:

- *loggers*
- *appenders*
- *layouts*
- *filters*

These four types are necessary to empower the developers with the capabilities to log messages according to message type and level, to control at runtime how these messages are formatted, and lastly to decide where they are reported.

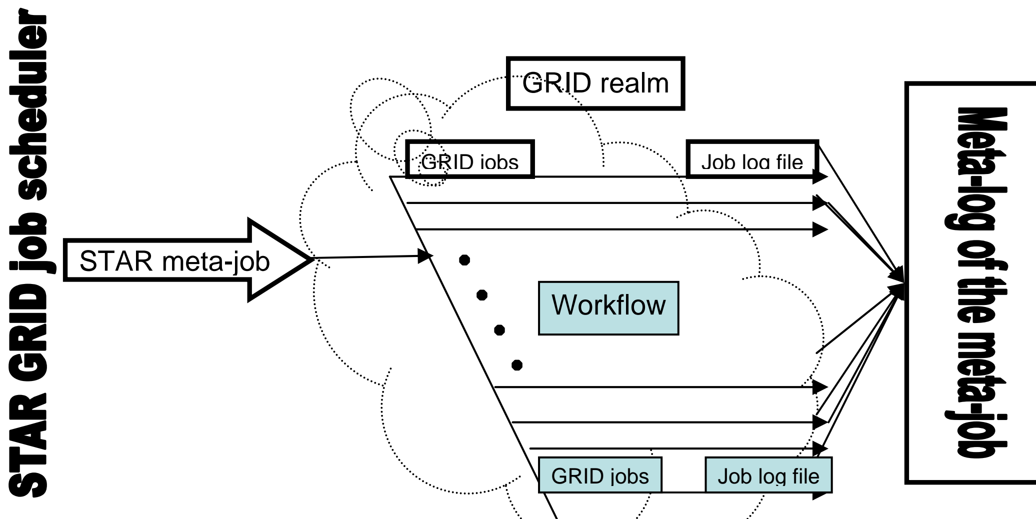

Figure 1: Example of a meta-job workflow for the STAR grid environment.

Run-time log configuration

Another advantage of using the log4j design [2] is the possibility to enable logging (or features of it) at runtime without modifying the application binary or the wrapper layers. Debugging statements can remain in shipped code without incurring a heavy performance cost. At the same time the logger equips the developer with context as detailed as necessary for understanding application failures, from an amount of information appropriate for testing or quality assurance, to a more limited set for production mode.

STAR LOGGER-BASED MESSENGER LAYER IMPLEMENTATION

Integration of the log4j packages and STAR offline framework was straightforward due to an already existing OO design for processing message streams [3]. Most (if not all) components of the STAR production framework send their log messages to the abstract STAR messenger.

To merge the STAR existing framework with the log4j package, it was only a matter of creating another implementation of the STAR messenger interface. Additionally, and to meet the STAR production needs, a standard set of the logger *appender* and *filter* features were either developed or enhanced. We added one extra *appender* to communicate with the STAR production MySQL database and created a custom logger *filter* to be able to select/suspend the messages according to the STAR production manager's criteria. Once again, and due to the already existing OO messenger implementation in the STAR production system, enabling the new scheme did not require modifying any existing piece of the STAR code. It was sufficient to dynamically load the new implementation and provide an external XML configuration file for the logger as needed.

CONCLUSION

The implementation of the STAR logger layer via log4cxx API has proven to be useful. It allows us to set independently the level of the verbosity for each STAR production "module". This is convenient for the debugging of the code as a whole as modules and classes are uniquely identified by their message format. In addition, each developer can set the message output levels for their own modules. The capability exists to include special *appenders* redirecting messages to a MySQL database or to HTML. This is particularly useful in certain critical cases where it allows us to get the job status on-line (via remote Db or Web page) before the GRID delivers the entire (usually large) log file to dig through. As needs for special treatment of the log or events in the chain of processing arise, a custom dynamically loaded *appender* filter can be developed and deployed, with no further need for code development or core application modification.

REFERENCES

[1] "Logging Services", http://logging.apache.org/.
[2] Ceki Gülcü, "Complete log4j Manual" , https://www.qos.ch/shop/products/log4j
[3] "StMessMgr: The STAR Offline Message Manager", http://www.star.bnl.gov/STAR/comp/sofi/StMessMgr/