ProtoLog: Qserv Logging Prototype built on Log4cxx

Features

- Simple, consistent interface across both C++ and Python (via swig).
- Multiple logging levels (trace, debug, info, warn, error, fatal).
- Multiple output channels (e.g. console, files, remote server).
- Format logging in C++ using either varargs/printf or `boost::format` style interfaces.
- Optionally retrieve logging object in C++ for high performance applications.
- Optionally include and format metadata including date, time, level, logger name, thread, function, source file, line number.
- Optionally add arbitrary thread-scoped metadata (e.g. key/value pairs) to logging context programmatically.
- Hierarchical, named loggers with independent configurations (e.g. levels, output channels, metadata, formatting).
- Dynamic hierarchical logging contexts via `push()` and `pop()` functions.
- Logging context objects automatically handle pushing/popping of names onto/off of global context in either C++ or Python.
- Offers alternative interface that is compatible with standard Python logging module.
- Configure loggers via single standard log4cxx XML or standard log4j Java properties file.
- Designate configuration file programmatically, via environment variable, or using default name and location.
- Automatically works without configuration file using default settings (e.g. logs to console).
- Programmatically adjust and query level thresholds.
- Use of macros in C++ allows conversion of logging/debugging statements into "no ops" in production versions of software.

Basic C++ examples

The varargs/printf style interface:

```
subject = "important stuff";
LOG("myLogger", LOG_LVL_INFO, "Here is some information about %s.", subject);
```

The `boost::format` style interface:

```
subject = "important stuff";
LOGF("myLogger", LOG_LVL_INFO, "Here is some information about %s." % subject);
```

Using the default logger:

```
LOG_DEBUG("My debugging statement.")
```

A logger object may be retrieved and used to avoid the cost of excessive lookups:

```
LOG_LOGGER logger = LOG_GET("myLogger");
LOG(logger, LOG_LVL_WARN, "Here is a warning sent using a logging object.");
```

Basic Python examples

```
    subject = "important stuff"
    protolog.log("myLogger", protolog.INFO, "Here is some information about %s.", subject)
```

```
    protolog.debug("My debugging statement.")
```

The standard Python `logging` module may also be used:

```
lgr = logging.getLogger()
lgr.setLevel(logging.INFO)
lgr.addHandler(protolog.ProtoLogHandler())
lgr.info("This is an info statement via the logging module.")
```

Example output

```
20140313 00:30:06,226 0x7f1f06bf6700 INFO  Here is some information about important stuff.
20140313 00:30:06,226 0x7f1f06bf6700 DEBUG My debugging statement.
```

```
357e700] INFO  myLogger qserv::master::AsyncQueryManager::add (bld/control/AsyncQueryManager.cc:149) - Here is
```

The first two examples above use the following formatting string to display the date, time, thread id, log level, and log message:

```
 "%d{yyyyMMdd HH:mm:ss,SSS} %t %-5p %m%n"
```

The last example above uses the following formatting string to display the date, time, thread id, log level, logger name, function, source file, source line number, and log message:

```
"%d [%t] %-5p %c{2} %M (%F:%L) - %m%n"
```

Logging Functions

In C++, the following varargs/printf style logging macros are available:

`LOG(loggername, level, fmt...)`
: Log a message of level **`level`** with format string **`fmt`** and corresponding comma-separated arguments to the logger named **`loggername`**.

`LOG_TRACE(fmt...)`
: Log a message of level `LOG_LVL_TRACE` with format string **`fmt`** and corresponding comma-separated arguments to the default logger.

`LOG_DEBUG(fmt...)`
: Log a message of level `LOG_LVL_DEBUG` with format string **`fmt`** and corresponding comma-separated arguments to the default logger.

`LOG_INFO(fmt...)`
: Log a message of level `LOG_LVL_INFO` with format string **`fmt`** and corresponding comma-separated arguments to the default logger.

`LOG_WARN(fmt...)`
: Log a message of level `LOG_LVL_WARN` with format string **`fmt`** and corresponding comma-separated arguments to the default logger.

`LOG_ERROR(fmt...)`
: Log a message of level `LOG_LVL_ERROR` with format string **`fmt`** and corresponding comma-separated arguments to the default logger.

`LOG_FATAL(fmt...)`
: Log a message of level `LOG_LVL_FATAL` with format string **`fmt`** and corresponding comma-separated arguments to the default logger.

In addition to the above macros, in C++, the following `boost::format` style logging macros are offered:

`LOGF(loggername, level, fmt)`
: Log a message of level **`level`** with format string **`fmt`** and corresponding `%` separated arguments to the logger named **`loggername`**.

`LOGF_TRACE(fmt)`
: Log a message of level `LOG_LVL_TRACE` with format string **`fmt`** and corresponding `%` separated arguments to the default logger.

`LOGF_DEBUG(fmt)`
: Log a message of level `LOG_LVL_DEBUG` with format string **`fmt`** and corresponding `%` separated arguments to the default logger.

`LOGF_INFO(fmt)`
: Log a message of level `LOG_LVL_INFO` with format string **`fmt`** and corresponding `%` separated arguments to the default logger.

`LOGF_WARN(fmt)`
: Log a message of level `LOG_LVL_WARN` with format string **`fmt`** and corresponding `%` separated arguments to the default logger.

`LOGF_ERROR(fmt)`
: Log a message of level `LOG_LVL_ERROR` with format string **`fmt`** and corresponding `%` separated arguments to the default logger.

`LOGF_FATAL(fmt)`
: Log a message of level `LOG_LVL_FATAL` with format string **`fmt`** and corresponding `%` separated arguments to the default logger.

In Python, the following logging functions are available in the `protolog` module. These functions take a variable number of arguments following a format string in the style of `printf()`. The use of `*args` is recommended over the use of the `%` operator so that formatting of log messages that do not meet the level threshold can be avoided.

`log(loggername, level, fmt, *args)`
: Log a message of level **`level`** with format string **`fmt`** and variable arguments **`*args`** to the logger named **`loggername`**.

`trace(fmt, *args)`
: Log a message of level `TRACE` with format string **`fmt`** and corresponding arguments **`*args`** to the default logger.

`debug(fmt, *args)`
: Log a message of level `DEBUG` with format string **`fmt`** and corresponding arguments **`*args`** to the default logger.

`info(fmt, *args)`
: Log a message of level `INFO` with format string **`fmt`** and corresponding arguments **`*args`** to the default logger.

`warn(fmt, *args)`
: Log a message of level `WARN` with format string **`fmt`** and corresponding arguments **`*args`** to the default logger.

`error(fmt, *args)`
: Log a message of level `ERROR` with format string **`fmt`** and corresponding arguments **`*args`** to the default logger.

`fatal(fmt, *args)`
: Log a message of level `FATAL` with format string **`fmt`** and corresponding arguments **`*args`** to the default logger.

Initialization

The underlying log4cxx system can be initialized from either the C++ or Python layers.

In C++, the following macros can be used:

`LOG_CONFIG(filename)`
: Initialize log4cxx using the XML or Java properties configuration file **`filename`** (see below).

`LOG_CONFIG()`
: Initialize log4cxx with default configuration file if found, otherwise using basic configuration (see below).

In Python, the following function is available in the `protolog` module:

`configure(filename)`
: Initialize log4cxx using the XML or Java properties configuration file **`filename`** (see below).

`configure()`
: Initialize log4cxx with default configuration file if found, otherwise using basic configuration (see below).

Configuration

The logging system is configured either using a standard log4cxx XML config file, a standard log4j Java properties, or using the default configuration. While log4cxx allows for programmatic configuration, as of this writing, only the adjustment of logging level threshold (see below) is exposed via the ProtoLog API. All other configuration (e.g. outputs and formatting) are controlled via a configuration file.

In the absence of an explicit call to one of the configuration macros, ProtoLog is designed to leverage log4cxx's static initializer, which employs the following algorithm (except from ?here):

1. Set the `configurationOptionStr` string variable to the value of the **LOG4CXX_CONFIGURATION** environment variable if set, otherwise the value of the **log4j.configuration** environment variable if set, otherwise the first of the following file names which exist in the current working directory, "log4cxx.xml", "log4cxx.properties", "log4j.xml" and "log4j.properties". If `configurationOptionStr` has not been set, then disable logging.
2. Unless a custom configurator is specified using the **LOG4CXX_CONFIGURATOR_CLASS** or **log4j.configuratorClass** environment variable, the PropertyConfigurator will be used to configure log4cxx unless the file name ends with the ".xml" extension, in which case the `DOMConfigurator` will be used. If a custom configurator is specified, the environment variable should contain a fully qualified class name of a class that implements the `Configurator` interface.

The C++ macro `LOG_CONFIG()` and the Python `protolog` module function `configure()`, which do not take a file path argument, utilize the above initialization algorithm with the following addition: Following

step (1), if the `configurationOptionStr` has not been set, log4cxx is configured using log4cxx's `BasicConfigurator`, which is hardwired to add to the root logger a `ConsoleAppender`. In this case, the output will be formatted using a `PatternLayout` set to the pattern "%-4r [%t] %-5p %c %x - %m%n".

Below is an example of an XML file that configures three appenders and two loggers: the root logger and the named logger "debugs". Each of the appenders contains a layout that defines which metadata to display and how to format log messages.

```
<?xml version="1.0" encoding="UTF-8" ?>
 <log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <!-- Output the log message to system console.
    -->
  <appender name="appxConsoleAppender" class="org.apache.log4j.ConsoleAppender">
    <param name="Target" value="System.out"/>
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%d{yyyyMMdd HH:mm:ss,SSS} %t %-5p %m%n"/>
        </layout>
  </appender>

  <!-- Output the log message to log file
    -->
  <appender name="appxFileAppender" class="org.apache.log4j.FileAppender">
    <param name="file" value="/u1/bchick/sandbox2/modules/var/log/qserv-master.proto.log" />
    <param name="append" value="true" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d %-5p %c{2} - %m%n" />
    </layout>
  </appender>

  <!-- My debugging file
    -->
  <appender name="debugsAppender" class="org.apache.log4j.FileAppender">
    <param name="file" value="/u1/bchick/sandbox2/modules/var/log/debugs.proto.log" />
    <param name="append" value="true" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d [%t] %-5p %c{2} %M (%F:%L) - %m%n" />
    </layout>
  </appender>

  <root>
        <priority value="info" />
        <appender-ref ref="appxFileAppender"/>
        <appender-ref ref="appxConsoleAppender"/>
  </root>


  <!-- Specify the level and appender for my logger -->
  <category name="debugs" additivity="false" >
        <priority value ="debug" />
        <appender-ref ref="debugsAppender"/>
  </category>

 </log4j:configuration>
```

The root logger is setup to append to both the console and the file `var/log/qserv-master.proto.log` with a threshold of "INFO". The named logger "debugs", meanwhile, is set to exclusively append to the file `var/log/debugs.proto.log` with a threshold of "DEBUG". In this way, "debugs" may capture and isolate verbose debugging messages without polluting the

main log. This behavior is triggered by setting the `additivity` attribute of the `category` tag to `"false"`. If `additivity` were set to `"true"` (the default value), all messages sent to "debugs" that met its threshold of "DEBUG" would also be sent to the console and `var/log/qserv-master.proto.log`, the targets of the appenders associated with the root logger.

Note that as an alternative to XML, a configuration file containing log4j Java properties may be used. Here's a trivial example of a log4j properties file that corresponds to the `BasicConfigurator` previously mentioned:

```
# Set root logger level to DEBUG and its only appender to A1.
log4j.rootLogger=DEBUG, A1

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout.
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```

Values in the configuration file can include Unix environment variables using the "`${ENVVAR}`" notation. In addition, "`${user.home}`" and "`${user.name}`" may be useful substitutions. Note that "~" substitution does *not* work.

Useful appenders include:

- `org.apache.log4j.ConsoleAppender`, taking a `target` property with values `System.out` and `System.err`
- `org.apache.log4j.FileAppender`, taking `file` and `append` properties
- `org.apache.log4j.rolling.RollingFileAppender`, taking a `rollingPolicy` property (but which can currently only be configured using an XML file, not a properties file)
- `org.apache.log4j.SyslogAppender`, taking `facility` and `syslogHost` properties

For the `PatternLayout`, all the conversion specifiers from ?this document should work.

Read more about log4cxx configuration ?here.

Programmatic Control of Threshold

The threshold level of any logger can be set or queried programmatically in both the C++ and Python layers.

In C++, the following macros are available:

`LOG_SET_LVL(loggername, level)`
: Assign **`level`** as the threshold of the logger named **`loggername`**.

`LOG_GET_LVL(loggername)`
: Returns the threshold level of the logger named **`loggername`**.

`LOG_CHECK_LVL(loggername, level)`
: Returns `true`/`false` indicating whether a logging message of level **`level`** meets the threshold associated with the logger named **`loggername`**.

`LOG_LVL_TRACE`
: Trace logging level (5000).

`LOG_LVL_DEBUG`
: Debug logging level (10000).

`LOG_LVL_INFO`
: Info logging level (20000).

`LOG_LVL_WARN`
: Warn logging level (30000).

`LOG_LVL_ERROR`
: Error logging level (40000).

`LOG_LVL_FATAL`
: Fatal logging level (50000).

In Python, the `protolog` module includes the following functions and variables:

`setLevel(loggername, level)`
: Assign **`level`** as the threshold of the logger named **`loggername`**.

`getLevel(loggername)`
: Returns the threshold level of the logger named **`loggername`**.

`isEnabledFor(loggername, level)`
: Returns `true`/`false` indicating whether a logging message of level **`level`** meets the threshold associated with the logger named **`loggername`**.

`TRACE`
: Trace logging level (5000).

`DEBUG`
: Debug logging level (10000).

`INFO`
: Info logging level (20000).

`WARN`
: Warn logging level (30000).

`ERROR`
: Error logging level (40000).

`FATAL`
: Fatal logging level (50000).

Logging Contexts

The logging context is maintained via a global stack onto which names can be pushed and popped using the ProtoLog macros `LOG_PUSHCTX(name)` and `LOG_POPCTX()` within C++ or the `protolog` functions `pushContext(name)` and `popContext()` within Python. The default logger name is dynamically constructed each time a name is pushed onto or popped from the logging context. For example, within the Qserv demo, the following code can be found within `app.py`:

```
protolog.initLog("/u1/bchick/sandbox2/modules/etc/Log4cxxConfig.xml")
protolog.pushContext("czar")
[...]
protolog.debug("This is a debug statement!")
```

The debugging statement is sent to the default logger, which in this case is *czar*. Later, within the `submitQuery3()` function of `control/dispatcher.cc`, we have:

```
LOG_PUSHCTX("control");
```

At this point, any logging statements that do not specify a logger name will be associated with a default logger of *czar.control*. This mechanism allows dynamic hierarchical logging without hardcoding full logger names in order to accommodate better code re-use. Logging could then be configured, for example, so that the *czar.control* context is redirected or copied to a special file with a unique logging level.

To pop a name from the logging context, invoke the `protolog.popContext()` function within Python or the `LOG_POPCTX()` macro within C++.

The context determines the default logger, which is used whenever a logger is not explicitly indicated or an empty string is provided as the logger name.

To summarize, within C++, we have the following macros:

`LOG_PUSHCTX(name)`
: Push **`name`** onto the global context stack such that the default logger name is appended with ".**`name`**".

`LOG_POPCTX()`
: Pop the last name `C` off of the global context stack such that a previous default logger name "`A.B.C`" is now "`A.B`".

`LOG_DEFAULT_NAME()`
: Returns the default logger name.

And in Python, the following functions are available in the `protolog` module:

`pushContext(name)`
: Push **`name`** onto the global context stack such that the default logger name is appended with ".**`name`**".

`popContext()`
: Pop the last name `C` off of the global context stack such that a previous default logger name "`A.B.C`" is now "`A.B`".

`getDefaultLoggerName()`
: Returns the default logger name.

Context Objects

Requiring that developers properly pop each context name is error prone. Therefore, we also provide logging context objects in both C++ and Python, which automatically pop names when the object is discarded.

In C++, the context object will pop the provided name in its destructor. Contexts can therefore be managed as follows:

```
{
    LOG_CTX context("demo");
    LOG_INFO("Info statement within demo context.");
}
LOG_INFO("Info statement outside demo context.");
```

In this example, if the context were initially "nameOne.nameTwo", the first logging statement would be sent to the logger named "nameOne.nameTwo.demo" and the second logging statement would be sent to the logger named "nameOne.nameTwo".

The LogContext class defined in the Python layer is compatible with Python's `with` statement, and therefore, the above example can be implemented in Python as follows:

```
    with ProtoLogContext(name="demo") as context:
        protolog.info("Info statement within demo context.")
    protolog.info("Info statement outside demo context.")
```

Fine-level Debugging Example

The following is a simple recipe for emulating additional debugging levels using the above API within Python. Analogous code can be readily written in C++ using the corresponding macros.

```
def debugLoggerName(num):
    """
    Returns the logger name that corresponds to fine-level debugging number NUM.
    """
    return '%s.debug%d' % (protolog.getDefaultLoggerName(), num)

def debugAt(num, fmt, *args):
    """
    Sends the log message created from FMT and *ARGS to the logger corresponding to fine-level debugging n
    """
    protolog.log(debugLoggerName(num), protolog.DEBUG, fmt, *args)

def debugSetAt(num):
    """
    Adjusts logging level thresholds to emulate debugging with fine-level NUM.
    """
    for i in range(5):
        protolog.setLevel(debugLoggerName(i), protolog.INFO if i < num else protolog.DEBUG)

debugSetAt(1)
debugAt(1, "Debug 1 statement that will display")
debugAt(2, "Debug 2 statement that will display")
debugAt(3, "Debug 3 statement that will display")
debugAt(4, "Debug 4 statement that will display")
debugAt(5, "Debug 5 statement that will display")
debugSetAt(2)
debugAt(1, "Debug 1 statement that will NOT display")
debugAt(2, "Debug 2 statement that will display")
debugAt(3, "Debug 3 statement that will display")
debugAt(4, "Debug 4 statement that will display")
debugAt(5, "Debug 5 statement that will display")
debugSetAt(3)
debugAt(1, "Debug 1 statement that will NOT display")
debugAt(2, "Debug 2 statement that will NOT display")
debugAt(3, "Debug 3 statement that will display")
debugAt(4, "Debug 4 statement that will display")
debugAt(5, "Debug 5 statement that will display")
debugSetAt(4)
debugAt(1, "Debug 1 statement that will NOT display")
debugAt(2, "Debug 2 statement that will NOT display")
debugAt(3, "Debug 3 statement that will NOT display")
debugAt(4, "Debug 4 statement that will display")
debugAt(5, "Debug 5 statement that will display")
debugSetAt(5)
debugAt(1, "Debug 1 statement that will NOT display")
debugAt(2, "Debug 2 statement that will NOT display")
debugAt(3, "Debug 3 statement that will NOT display")
debugAt(4, "Debug 4 statement that will NOT display")
debugAt(5, "Debug 5 statement that will display")
```

Mapped Diagnostic Context

User-specified metadata in the form of key/value pairs may be given to ProtoLog using the macro `LOG_MDC(key, value)` within C++, or the `protolog` function `MDC(key, value)` within Python. These metadata may then be automatically included as part of any/all subsequent log messages by specifying the corresponding key in the appender's formatting string as per standard log4cxx. These metadata are handled using log4cxx's mapped diagnostic context (MDC) feature, which has thread-level scope.

For example, a session id may be included by using the following formatting string:

```
"%d [%t] %-5p %c{2} (%X{session}) %m%n"
```

In C++, the following MDC macros are available:

`LOG_MDC(key, value)`
: Map the value **`value`** to the global key **`key`** such that it may be included in subsequent log messages by including the directive `%X{`**`key`**`}` in the formatting string of an associated appender.

`LOG_MDC_REMOVE(key)`
: Delete the existing value from the global map that is associated with **`key`**.

In Python, the `protolog` module provides the following MDC functions:

`MDC(key, value)`
: Map the value **`value`** to the global key **`key`** such that it may be included in subsequent log messages by including the directive `%X{`**`key`**`}` in the formatting string of an associated appender. Note that `value` is converted to a string by Python before it is stored in the MDC.

`MDCRemove(key)`
: Delete the existing value from the global map that is associated with **`key`**.

All key/value pairs in the MDC can be included in a single log line using a plain `%X` directive (without a key). Each of the key/value pairs will be separated by a comma and enclosed in braces; the entire list is enclosed in another set of braces.

Benchmarks

Measuring the performance of ProtoLog when actually writing log messages to output targets such as a file or socket provides little to no information due to buffering and the fact that in the absence of buffering these operations are I/O limited. Conversely, timing calls to log functions when the level threshold is not met is quite valuable since an ideal logging system would add no appreciable overhead when deactivated. Basic measurements of the performance of ProtoLog have been made with the level threshold such that logging messages are not written. These measurements are made within a single-node instance of Qserv running on lsst-dev03 without significant competition from other system activity. The average time required to submit the following suppressed log message is 26 nanoseconds:

```
    LOG_INFO("Hello default logger!");
```

The same holds for the `boost::format` style interface (also 26 nanoseconds):

```
    LOGF_INFO("Hello default logger!");
```

Importantly, the overhead is unaffected when formatting is introduced since the CPU will not encounter these instructions when the logging threshold is not met. For example, the average time required to submit the following suppressed log message is also 26 nanoseconds:

```
const char* info_stuff = "important stuff";
LOGF_INFO("Hello default logger with %s." % info_stuff);
```

Note that these timings are indistinguishable from those using log4cxx directly, without the ProtoLog layer.

Meanwhile, the situation is quite different when looking up the logger by name. For instance, the average time required to submit the following suppressed log message is 1.0 microseconds (~40X slower):

```
LOG("myLogger", LOG_LVL_INFO, "Hello my logger!");
```

The same holds when using the `boost::format` style interface or when formatting. That is, the following the example also takes 1.0 microseconds:

```
const char* info_stuff = "important stuff";
LOGF("myLogger", LOG_LVL_INFO, "Hello my logger with %s." % info_stuff);
```

As an alternative to using the default logger, intermediate performance can be achieved for suppressed log messages using logger objects. Retrieve the logger object by name:

```
LOG_LOGGER myLogger = LOG_GET("myLogger");
```

Now, the following suppressed log message takes 190 nanoseconds:

```
LOG(myLogger, LOG_LVL_INFO, "Hello my logger!");
```

The 190 - 26 = 164 nanoseconds differential between this and the `LOG_INFO` example above is attributable to looking up the logging level `LOG_LVL_INFO`.

These measurement were made using the OpenMP API with code like the following:

```
int iterations = 1000;
double start, stop;
LOG_SET_LVL("", LOG_LVL_WARN);
assert(!LOG_CHECK_LVL("", LOG_LVL_INFO));
start = omp_get_wtime();
for (int i=0; i < iterations; i++)
    // These log messages will not print.
    LOG_INFO("Hello default logger!");
stop = omp_get_wtime();
LOG_WARN("LOG_INFO(...): avg time = %f" % ((stop - start)/iterations));
```

Source Code

Source code for the logger can be found in the git repository ?here.