Lat revision: 17 November 2010

Logging Best Practices Guide

Dan Gunter
Lawrence Berkeley National Laboratory

Introduction

Real-time status outputs from running software, which we will here call "logs", are instrumental to performance analysis, problem diagnosis, and security auditing tasks such as incident tracing and damage assessment. Too often, though, logs are only useful to the author of the program, poorly structured, or missing important information. The solution to this should be obvious: logs need to be well-structured and complete. If we apply the same discipline and consistency to logging as we apply to coding standards, logs can be readily useful to the wider community of administrators, users, and middleware developers (and the log-consuming programs that all these people create). The purpose of this document is to describe the essential elements of that discipline, and recommend specific best practices for each of those elements.

This document focuses on the content of the logs and does not discuss logging APIs. Whether you use a standard log API or *printf* is your business, although large programs will benefit from a single set of specialized logging functions that enforce consistency. [As an aside, our software NetLogger (http://acs.lbl.gov/NetLoggerWiki), contains log formatting and generation functions in a number of languages that follow the best practices described herein.]

The Best Practices format is similar in many ways, and certainly compatible in spirit, with Spunk's "Common Information Model" (see http://www.splunk.com/wiki/Apps:Common_Information_Model).

What should be logged?

In general, anything that may take a long time or fail should be logged. Wherever possible *both the beginning and the end of an operation should be logged separately*. The basic reason for doing this, rather than logging the operation once at the end, is that a fatal error or stall is then much easier to trace back to the "last thing I started". Some will object that this practice doubles the log volume, but this can be compensated for in two ways: first, the level of detail is a much more effective way of regulating log volume; second, a relatively simple log consumer can be inserted downstream that combines completed operations into a single event.

Some specific examples of things that should be logged are:

- **Service initiation, configuration and termination**: Whenever a service starts up or a service request thread is launched, this should be logged. If the service can be configured, this message must contain a reference to the service configuration used. The termination message should include a status or termination message or code.
- **Errors**: All errors that cause a component to exit should be logged.
- **Authentication and authorization operations**: Authentication events should include the authentication method and claimed identity. Authorization events should include the remote identity, mechanism, and mechanism-specific attributes.
- **Entering or exiting** functions and major loops

How should events be logged?

The essential elements for constructing high-quality logs are as follows:

- **Structure and format**: Consistently structured, self-describing, ASCII text records
- **Level of detail**: Use of logging levels to separate logs by detail
- **Event types**: Unique names for each logged event from a hierarchical namespace
- **Timestamps**: High-resolution timestamps in a standard format
- **Identifiers**: Explicit and clearly labeled identifiers for resources

The remainder of this document provides some common terminology, then goes on to discuss each of these elements in more detail.

Terminology

The terminology used in this document distinguishes between the abstract information model and the (serialized) representation of that model. An *event* and *attribute* are part of the abstract model, and the remaining terms belong to the representation.

event - Point of interest within a given system occurring at a specific time. Events are the things your logs are communicating to the outside world.

attribute - Detailed characteristic of an event, which has a name and value

log - Representation of a stream of events

log record - Representation of a single event within a log *keyword/value pair* - Representation of the name and value of an attribute

Structure and format

Consistently structured, self-describing, ASCII text records

Structure

Every log record should have the same structure and format. This structure includes standard attributes for all the essential elements described in the introduction, plus additional attributes specific to the event. To achieve this, the recommended log record structure is a set of keyword/value pairs, also called an associative array, where each pair represents an attribute's name and value.

Format

The recommended log record format is a newline-terminated line of 7-bit ASCII text, where each keyword/value pair is connected by an equals sign and separated from other pairs by whitespace. The order of attributes does not matter.

Keyword strings: Any combination of alphanumeric 7-bit ASCII characters plus dot, underscore and dash. Keywords should be kept short, if possible, and no longer than 128 characters.

Value strings: Value strings can be surrounded with double-quotes or be unquoted. If unquoted, they can be any combination of printable non-whitespace 7-bit ASCII characters up to 255 characters long. Values with whitespace must be quoted, and embedded quote characters escaped with a backslash.

Timestamp: For details on the timestamp format, see the **Timestamps** section.

For example here are two log records describing the start and end of the "doit" process:

```
ts=2006-12-08T18:39:19.372375Z level=INFO event=doit.start

ts=2006-12-08T18:39:22.820440Z level=INFO event=doit.end status=0
```

Note that each log record is self-describing, at the price of repetition of the attribute keywords. Some may be tempted to find a way to remove the repeated keywords by ordering the attributes and adopting a header/body distinction. But in the long run, the inefficiencies of this format are offset by the advantages of each log event being easily processed in isolation: logs can be reordered, broken up, and reassembled without any loss of information. In the example above, "doit.end" could be arbitrarily separated from "doit.start" in the stream without changing the meaning. In complex multi-component systems, this property greatly simplifies the processing of end-to-end logs.

As a further example, here is how a log record with values embedded in an English phrase could be transformed into the recommended format (with line breaks added to fit the page width).

Original

```
error: read from socket on foobar.org:1234, remote host baz.org:4321
returned -1
```

Best-practices

```
ts=2006-12-08T18:48:27.598448Z event=socket.read level=ERROR status=-1
host.local=foobar.org:1234 host.remote=baz.org:4321
```

Newline-termination, although somewhat inconvenient for stack traces, is compatible with UNIX *syslog* and related infrastructure such as *syslog-ng* and *rsyslog*, as well as being a natural input format for most text processing tools (e.g., UNIX *grep*).

Standard keywords

In order to easily process logs from varying applications, it is helpful to have reserved keywords for standard cross-cutting elements of a log record. The following keywords and keyword patterns are recommended. In the patterns, "*" is used as a wildcard meaning one or more characters.

- **`ts`** – Time of event: See the *Timestamps* section.
- **`event`** - Type of event: See the *Event types* section.
- **`level`** - LOD of event: See the *Level of detail* section.
- **`status`** - Status of an operation: With event values Integer value with 0 (zero) indicating success, negative values indicating failures, and positive values indicating something in between.
- **`guid`** or **`*.id`** - Identifiers: See the *Identifiers* section.
- **`*.start`**, **`*.end`**, **`*.error`** - Start, end, or error in some operation. For `.end`, the `status` keyword should be used to indicate the return code.

Level of Detail (LOD)

Use of logging levels to separate logs by detail

Logging should, in theory, record each significant event in the system. In practice this can result in far more logging detail than most log consumers want. Some filtering can be done before records are emitted, but often a superset of desired detail must be output first, then filtered by consumers. To allow the infrastructure to easily and efficiently filter log records based on their content, each log record should have a level of detail (LOD) attribute, represented by the `level` keyword. This same LOD also distinguishes between errors and informational messages. A recommended set of values for the LOD can is a combination of the syslog and Apache levels, as follows:

- FATAL: Component cannot continue, or system is unusable.
- ALERT: Action must be taken immediately.
- CRITICAL: Critical conditions (on the system).
- ERROR: Errors in the component; <u>not</u> errors from elsewhere.
- WARNING: Problems that are recovered from, usually.
- NOTICE: Normal but significant condition.
- INFO: Informational messages that would be useful to a deployer or administrator.
- DEBUG: Lower level information concerning program logic decisions, internal state, etc.
- TRACE: Finest granularity, similar to “stepping through” the component or system.

If there is no LOD in the log record, a consumer should that it is informational (INFO).

Event types

Unique names for each logged event from a hierarchical namespace

Each event has a type, represented by the special `event` keyword. The event type could be anything, but the recommended practice is to arrange the the types of objects and processes it represents into a hierarchy and then concatenate them in event value with a '.' (period) between each component, starting at the most abstract. For all the usual reasons, it is good practice to prefix all the types for a given program with a common namespace. A good choice for this namespace is the reversed DNS name of the project (as is the common practice for Java class namespaces), such as "com.google".

The particular style of concatenation is a matter of preference -- consistency is best, but remember that we are replacing descriptive phrases, and almost anything is better than arbitrary English grammar rules. For example, let's say we want to represent a single strike in a baseball game. The event types for these three events could be stated in the pattern *(namespace).(actor).(action)*: mlb.batter.raise_bat, mlb.pitcher.windup, mlb.pitcher.throw, mlb.batter.swing, mlb.catcher.catch.

Timestamps

High-resolution timestamps in a standard format

Timestamps are a perennial source of headaches in any kind of distributed system, with the most common problem being lack of timezone information.

There are two recommended timestamp formats, one that's more human-friendly and one that's easier for machines. The recommended human-friendly format is a legal variant of the ISO8601 time standard, with separator characters to make it easier to scan:

YYYY-*MM*-*DD*T*HH*:*MM*:*SS*.*SSSSSS*Z

For example, October 26th, 2000 at 8:34 and 26.30323 seconds UTC would be: 2000-10-26T08:34:26.30323Z. The “Z” at the end signifies UTC time. This is preferred, but a positive or negative offset may replace the "Z", in the format "+/-dddd", e.g. "-0700".

The more machine-friendly format is the number of seconds since January 1, 1970 (the "UNIX epoch"). For example the time given above, 2000-10-26T08:34:26.30323Z, would be 972549266.30323.

In either case, microsecond timestamp resolution is recommended even where it's not obviously needed, on the theory that it's much easier to ignore extra digits than to guess them.

Identifiers

Explicit and clearly labeled identifiers for resources

In programs with parallel or many serial activities, or those composed of many components -- in other words, most programs -- it is important to use explicit identifiers that allow log consumers to to link log records together into streams of activity without depending too heavily on the details of the log record contents. Recall that the log record syntax supports this by reserving the attribute keyword `guid` and any keyword ending in `.id` for identifiers. These identifiers should, of course, be repeated in all log records that belong to the same stream(s) of activity.

The value of an identifier is not constrained, but it is recommended that Global Unique Identifiers (GUIDs) be used where reasonable to allow the identifier to be used as-is in any scope. These could be used in addition to more "natural" identifiers that are tied to program semantics, such as a request, process, or thread identifier.

For example, the simple series of actions involved in a baseball pitch (see the **Event type** section) might all be linked together with a 'game.id' as well as the more natural inning and batter-this-inning numbers, while individual actions might be associated with players through a 'player.id'.

Concluding remarks

If there was one phrase to sum up this document, it would be "Structured and consistent logging makes programs easier to debug and maintain." The same could be said for a good test suite and documentation, and in fact all three of these work together.

One issue frequently arises in interactive programs. Warnings and errors need to be shown to the user, and the very structured and complete log format shown above is not easy for people to read and understand. The recommended solution to this problem is to have two streams of messages emitted by the program: one for human consumption and another, structured one, for debugging and performance analysis. For many programs, it is also useful to have a "batch mode" that can make all the logs be structured. An alternate solution is to simply change the *format* of the messages in "interactive" mode, while retaining the same structure. This is a little less friendly to the user (the message will not be as readable) but allows for just one set of log statements in the program.

Contact

If you wish to contact the author of this document, you can send email to: dkgunter@lbl.gov

Feedback and suggestions for improvement are always appreciated.