Log++ Logging for a Cloud-Native World

Mark Marron
Microsoft Research
Microsoft
USA
marron@microsoft.com

Abstract

Logging is a fundamental part of the software development and deployment lifecycle but logging support is often provided as an afterthought via limited library APIs or third-party modules. Given the critical nature of logging in modern cloud, mobile, and IoT development workflows, the unique needs of the APIs involved, and the opportunities for optimization using semantic knowledge, we argue logging should be included as a central part of the language and runtime designs. This paper presents a rethinking of the logger for modern *cloud-native* workflows.

Based on a set of design principles for modern logging we build a logging system, that supports near zero-cost for disabled log statements, low cost lazy-copying for enabled log statements, selective persistence of logging output, unified control of logging output across different libraries, and DevOps integration for use with modern cloud-based deployments. To evaluate these concepts we implemented the Log++ logger for Node.js hosted JavaScript applications.

CCS Concepts • **Software and its engineering → Compilers**; **Runtime environments**; *General programming languages*;

Keywords Logging, JavaScript, Runtime Monitoring

ACM Reference Format:
Mark Marron. 2018. Log++ Logging for a Cloud-Native World. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '18), November 6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3276945.3276952

DLS '18, November 6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6030-2/18/11…$15.00
https://doi.org/10.1145/3276945.3276952

1 Introduction

Logging has always been a important tool for software developers to gain understanding into their applications [25, 30, 32]. However, as DevOps oriented workflows have become more prevalent, logging is becoming an even larger consideration when building applications [11, 32]. A key area driving this shift is the use of cloud-based applications and the integration of application monitoring dashboards, such as Stack Driver [27], N|Solid [21], or AppInsights [1], which ingest logs from an application, correlate this information with other aspects of the system, and provide the results in a friendly dashboard format for developers. The additional value provided by these dashboards and the ability to quickly act on this data makes the inclusion of rich logging data an integral part of application development.

Existing logging library implementations, as provided via core or third party libraries, are unable to satisfactorily meet the demands of logging in modern applications. As a result developers must use these loggers with care to limit undesirable performance impacts [34] and *log spew* [11, 34], work to control logging output from other modules to the appropriate channels, and figure out how to effectively parse the data that is written from various sources. Consider the JavaScript code in Figure 1 which illustrates concrete issues encountered by Node.js [19] developers today.

A major issue with logging is the potential for the accidental introduction of serious performance problems though seemingly benign activities [11, 25, 30, 32, 34]. In existing logging frameworks even when a logging level is disabled, as `debug` and `trace` levels usually are, the code to generate and format the log message is still executed. This can either be due to eager evaluation semantics of the source language or due to limitations in compiler optimizations for dead-code elimination in languages with workarounds such as macros. This results in code that looks like it will not be executed but that, in reality, incurs large parasitic costs as can be seen in the `logger.debug` statement in the example, which at the default level does not print to the log, but will still result in the creation of the literal object and generation of a format string on every execution of the loop. This cost leads developers to defensively remove these statements from code instead of depending on the runtime to eliminate their costs when deploying an application.

Next is the issue of *log spew* [11, 34] where logging at a detailed level, which may be desirable for debugging when

```
1  const dest = fs.createWriteStream("/tmp/logging/app.log");
2  const logger = require('pino')(dest, { level: "info" });
3
4  function foo(name, flag) {
5      console.log("Hello");
6      logger.info("World");
7      for (var i = 0; i < 1000; ++i) {
8          logger.debug("Data = " + JSON.stringify({ nval: name, cval: i }));
9          ...
10     }
11
12     const ok = check(name, flag);
13     logger.info("check(%s, %b) is %b)", name, flag, ok);
14     if (!ok) {
15         logger.warn("Error ...");
16     }
17 }
```

Figure 1. Examples of logging usage in JavaScript

an issue occurs, fills the log with large quantities of uninteresting noise output. An example of this is the `logger.info` message about the args and result of the `check` call in Figure 1. In the case of a successful execution the content of this log statement is not interesting and the cost of producing it plus the increased noise in the logfile is pure overhead. However, if the `check` statement fails then having information about what events led up to the failure may be critical in diagnosing/fixing the issue. In current logging frameworks this is an unavoidable conundrum and, in any case where the trace history is needed, the logging statements must be added and the cost/noise accepted as a cost.

The combination of verbose logging and the trend towards including critical, but extensive, metadata such as timestamps and host information in log messages further drives concerns about the performance of logging. Computing a timestamp or a hostname string is inexpensive but the cost of formatting them into a message is non-trivial can can add up over thousands or millions of log messages resulting in unexpected performance degradation.

Modern developer practices around logging frequently involve post processing of log data into analysis frameworks like the Elastic stack [7] or Splunk [26]. However, free form specification of message formats, as seen in `printf` or concatenated value styles, are not amenable to machine parsing. Modern logging frameworks, log4j [13], pino [23], bunyan [3], etc. provide some support for consistently formatting and structuring output but fundamentally this problem is left as a problem development teams need to solve via coding conventions and reviews.

The final issue we consider is the growing pain of integrating multiple software modules, each of which may use a different form of logging. In our running example we have `console.log` writing to the `stdout` and a popular framework called `pino` which has been configured to write to a file. As a result some log output will appear on the console while other output will end up in a file. Further, if a developer changes the logging output level for `pino`, from say `info` to `warn`, this will not change the output level of the `console` output. Developers can work around this to some degree by enforcing the use of a single logging framework for their code but they will not always be able to control the frameworks used by external libraries.

To address these issues we propose an approach where logging is viewed as a first class feature in the design/implementation of a programming language and runtime instead of simply another library to be included. Taking this view enables us to leverage language semantics, focused compiler optimizations, and semantic knowledge in the runtime to provide a uniform and high performance logging API.

The contributions of this paper include:

- The view that logging is a fundamental aspect of programming and should be included as a first class part of language, compiler, and runtime design.
- A novel dual-level approach to log generation and writing that allows a programmer to log execution data eagerly but only pay the cost of writing it to the log if it turns out to be interesting/relevant.
- Using this dual-level approach we show how to separate and support the desire to use logging for both debugging when an error condition is encountered and for telemetry purposes to monitor general application behavior.

- A suite of innovative log format and log level management techniques that provide a consistent and unified log output that is easy to manage and feed into other tooling.
- An implementation in Node.js to demonstrate that key ideas can be applied to existing languages/runtimes and to provide an production quality implementation for use in performance evaluations.

2 Design

This section describes opportunities, using language, runtime, or compiler support, to address general challenges surrounding logging outlined in Section 1. We can roughly divide these into two classes – performance oriented and functionality oriented.

2.1 Logging Performance

Design Principle 1. *The cost of a logging statement at a logging level that is disabled should have zero-cost at runtime. This includes both the direct cost of the logging action and the indirect cost of of building a format string and processing any arguments. Further, disabling/enabling a log statement should not change the semantics the application.*

When logging frameworks are included as libraries the compiler/JIT does not, in general, have any deep understanding of the enabled/disabled semantics of the logger. As a result the compiler/JIT will not be able to fully eliminate dead-code associated with disabled logging statements and will incur, individually small but widespread, parasitic costs for each disabled logging statement. These costs can be very difficult to diagnose, as they are widely dispersed and individually small, but can add up to several percentage points of application runtime. To avoid these parasitic costs we propose including logging primitives in the core specification of the programming language or, if that is not possible, adding compiler/JIT specializations to optimize them.

An additional advantage of lifting log semantics to the language specification level is the ability to statically verify logging uses and handle errors that occur during log expression evaluation. Common errors include format specifier violations [28], accidental state modification [5, 16] in the logging message computation, and other logging anti-patterns [4]. If the language semantics specify logging API's then both of these error classes can be statically checked to avoid runtime errors or heisenbugs that appear/disappear when logging levels are changed.

Design Principle 2. *The cost of an enabled logging statement has two components – (1) the cost to compute the set of arguments to the log statement and (2) the cost to format and write this data into the log. The cost of computing the argument values is, in general unavoidable, and must be done on the hot path of execution. However, the cost of (2) should be reduced and/or moved off the hot path as much as possible.*

To minimize the cost of computing arguments to the log statement and speed their processing we propose a novel log format specification mechanism using *preprocessed* and stored log formats along with a set of *log expandos* which can be used as a shorthand in a log to specify common, but expensive/complicated, to compute log argument values. The use of preprocessed format messages allows us to save time, the type checking and processing of each argument does not require parsing the format string, and instead of eagerly stringifying each parameter we can do a quick immutable copy of the argument which can be formatted later. Expandos provide convenient ways to add data into the log, such as the current date/time, the host IP, or a current request ID [14], that would either be more expensive or more awkward to compute explicitly on a regular basis. We can eliminate the main-thread cost of formatting by batching the log messages and instead doing the format work on a background thread.

2.2 Logging Functionality

Design Principle 3. *Logging serves two related, but somewhat conflicting roles, in modern systems. The first role is to provide detailed information on the sequence of events preceding a bug to aid the developer in triaging and reproducing the issue. The second role is to provide general telemetry information and visibility into the overall behavior of the application. This observation leads to the third design principle: logging should support both tasks simultaneously without compromising the effectiveness of either one.*

To support these distinct roles we propose a dual-level logging approach. In the first level all messages are initially stored, as a format + immutable arguments, into an in-memory buffer. This operation is high performance and suitable for high frequency writes of detailed logging information needed for debugging. Further, in event an error is encountered the full contents of detailed logging can be flushed to aid in debugging. In the second level these detailed messages can be filtered out and only the high-level telemetry focused messages can be saved, formatted, and written into the stable log. This filtering avoids the pollution the saved logs with overly detailed information while preserving the needed data for monitoring the overall status of the application [11, 32].

Design Principle 4. *Logging code should not obscure the logic of the application that it is supporting. Thus, a logger should provide specialized logging primitives that cover common cases, such as conditional logging, that would otherwise require a developer to add new logic flow into their application specifically for logging purposes.*

Common scenarios that often involve additional control or data flow logic include *conditional logging* where a message is only written when a specific condition is satisfied, *child loggers* which handle a specific subtask and often developers

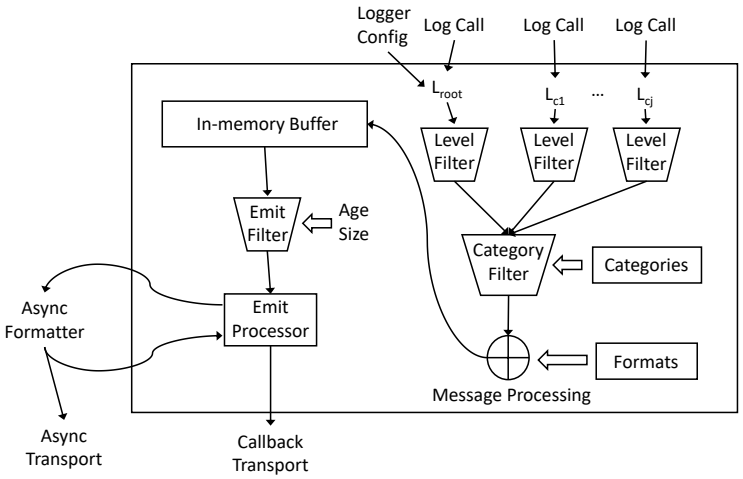

Figure 2. Logger architecture

want to include additional information in all log messages from this subtask, and *bracketing entries* where developers want to mark the start/end of something and include correlated timing (and other) information in the bracketing. All of these scenarios involve the developer adding additional, error-prone, control and data flow to the program which obscures the core algorithmic code. Thus, we propose adding primitive methods for supporting all of these scenarios without requiring additional developer implemented logic.

Design Principle 5. *Applications are often built leveraging numerous third-party components which may include their own logging functionality and may try to configure their own output endpoints. A logging framework should provide a mechanism for composing log outputs and controlling logging related configurations of included modules and components.*

Supporting this workflow requires that the language and/or runtime to standardize a number of features. The first is the set of logging levels and way to define categories. Without these it is not even possible for the loggers to agree on what is expected to be output. The next is a unified way to aggregate and output messages to a common destination. Finally, since the toplevel application needs to have the final authority over which levels/categories are enabled for which modules and where the data goes, a logger must have the concept of a *root logger* which can modify these values and *sub-loggers* from included modules which should behave in accordance with the specifications provided by the root logger.

3 Implementation

Given the design principles outlined in Section 3 we now present the implementation of Log++[1] which realizes these goals in a logger for the `Node.js` [19] runtime. It is possible to implement many of the features needed to satisfy our design goals as a library or using the native API extension bindings (N-API [18]) but others require core runtime support. For these core changes we modify the ChakraCore JavaScript engine and core Node implementation directly.

3.1 Implementation Overview

The logging system is split into five major components that (1) manage the global logger states, message filters, message formats, and configurations (2) the message processor and in-memory buffer (3) the emit filter and processor (4) the formatter (5) and finally the transports. These components and the relations between them are shown in Figure 2 and explained in detail in the rest of the section.

3.2 JavaScript Implementation

Log State Manager The first component we look at in the implementation is the global log state manager. This component is responsible for tracking all of the loggers that have been created, which one of them (if any) is the root logger, the enabled logging levels + categories, and the message formats which have been defined. The loggers are shown as L_{root} and $L_{c1} \ldots L_{cj}$ in Figure 2 and each one is associated with a Level Filter that controls which levels are enabled/disabled for it. The global category filter controls which categories are enabled/disabled on a global basis.

As seen in the example code there be many loggers created in different parts of the application. One logger, named "app", is created on line 2 of the main application in Figure 3 while a second, named "foo", is created in the module `foo.js` in Figure 4 which is included from the main `app.js` file. As stated in *design principle 5* we do not want the included sub-module `foo.js` to be able to, unexpectedly, change the

[1] Log++ sources available at https://github.com/mrkmarron/logpp

```
1  //app.js
2  const logapp = require("logpp")("app");
3  logapp.addFormat("Hello", "Hello %s %j");
4
5  const foo = require("./foo");
6  foo.doit();
7
8  logapp.info(logapp.$Hello, "info", {f: 3});
9  logapp.detail(logapp.$Hello, "detail", {f: 4});
10
11 logapp.addFormat("Now", "#walltime");
12 logapp.info(logapp.$$Perf, logapp.$Now);
13 logapp.enableCategory("Perf");
14 logapp.info(logapp.$$Perf, logapp.$Now);
```

Figure 3. Running example main app code

```
1  //foo.js
2  const logfoo = require("logpp")("foo");
3  logfoo.addFormat("Hello2", "Hello2 %s");
4
5  function doit() {
6      logfoo.info(logfoo.$Hello2, "foo.js");
7  }
8
9  logfoo.setOutputLevel(logfoo.Levels.TRACE);
10 module.exports.doit = doit;
```

Figure 4. Running example submodule code

logging level for the main app (the call to `setOutputLevel` on line 9). We also allow the root logger to enable/disable log outputs from these subloggers.

To support these features the Log++ logger keeps a list of all created loggers and a special *root logger* which is the first logger created in the main module of the application. When updating logging levels or creating a new logger we check if the action is coming from the root logger and, if not, either convert the call to a `nop` or look at the root logger configuration to see if we need to override any parameters. These features cover the needs of controlling logging from multiple modules as described in *Design Principle 5*.

In our running example the state manager will intercept the creation of the logger on line 2 in `foo.js`. Since the logger created here is not the root logger we intercept this construction, set the in-memory log level to the overridden (default) `WARN` level instead of the standard value of `DETAIL`, prevent the modification of the *emit-level* on line 9, and will store the newly created logger in a list of subloggers.

Tracking the list of all created sub-loggers allows a developer to share a single logger between several files in the same module. The name parameter in the logger constructor is keyed with the logger and, if the same key is used in multiple places, the same logger object will be returned.

Finally, the state manager is responsible for maintaining information on the current *emit levels*, the enabled/disabled categories, and the sublogger override information. Each logger has an independent level at which it can write into the in-memory log. However, the state manager maintains a global set of enabled/disabled categories for all loggers and a global logging level for eventual emit processing, which only the root logger is allowed to update.

In our running example the main application has a log statement on lines 12 and 14 in the `Perf` category to put a message with the current walltime in the log. The first log statement happens before the `Perf` category has been enabled it will not be processed. After enabling the `Perf` category on line 13 the log operation on line 14 will be processed and results in a message saved to the in-memory log.

Message Formats To improve performance and support machine parsing of the log output we adopt a *semantic logging* approach where the log call copies the format information and message arguments to a secondary location instead of formatting them immediately. This implementation choice supports the needs of *Design Principle 2*. Since the copy operation is low cost this minimizes the performance impact on the main thread and allows the formatter to build up a parser for all the messages it emits which can be later used to parse the log. Modern software development also favors the use of consistent styles and data values in the log. Thus, Log++ encourages developers to split the logging action into two components:

- Format definition using an `addFormat` method which takes a format string or JSON format object, processes it into an optimized representation, and then saves it for later use.
- Use a previously defined format in a log statement by passing the previously generated format identifier and the list of arguments for processing.

In addition to programmatically processing single log formats, as shown on line 3 in Figure 3, we also allow the programmer to load formats in bulk from a JSON formatted file. This allows a team to have a unified set of logging messages that can be loaded/processed quickly on startup and then used repeatedly throughout the applications execution. Once loaded all format objects are saved in the *Formats* component in Figure 2 where they can be loaded as needed when processing a log statement.

The format for a logger message is:

```
1  {
2    formatName: string, //name of the format
3    formatString: string, //raw format string
4    formatterEntries: {
5      kind: number, //tag indicating the entry kind
6      argPosition?: number, //position in argument list
7      expandDepth?: number, //JSON expand depth
8      expandLength?: number //JSON expand length
9    }[]
10 }
```

This representation allows us to quickly scan and process arguments to a log statement as described in the *In-Memory Message Processing* section. The kind information is used for both identifying what type of value is expected when formatting an argument, e.g. `number`, `string`, etc., but we also use it to support *format macros*.

To support the easy/efficient logging on a number of common values that are not easily (or cheaply) computable we provide *format macros*. Classic examples include adding the current walltime or the hostname as part of a log message. In JavaScript these require explicitly calling expensive compound API sequences `new Date().toISOString()` or `require('os').hostname()`. Instead we allow a developer to use special macros `#walltime`, as seen on line 11 in Figure 3, or `#host` in their format strings and then the logger has optimized internal paths to get the needed values. In these cases the `kind` field is set to the enumeration for the macro and the `argPosition` is undefined. The list of supported macro formatters includes:

- #host – name of the host
- #app – name of the root application
- #logger – name of the logger
- #source – source location of log statment (file, line)
- #wallclock – wallclock timestamp (ISO formatted)
- #timestamp – logical timestamp
- #request – the current request id (for http requests)

A common logging practice is to include raw objects, formatted as JSON, into the message. This is a convenient way to include rich data in a message but can lead to unexpectedly excessive logging loads when, what the developer expected to be a small object, turns out to be quite large. To prevent this we have a specialized JSON-style processor that will bound the depth/length of object expansion during formatting. The `expandDepth` and `expandLength` arguments provide control over this depth/length and can be adjusted in the format string when a developer want to capture more (or less) information than what is provided by the defaults.

In-Memory Message Processing The in-memory buffer is implemented as a linked-list of block structures:

```
1  {
2    tags: Uint8Array,
3    data: Float64Array,
4    strings: string[],
5    properties: map<number, string>
6  }
```

To allow efficient marshalling of the data from our JavaScript logger frontend to the C++ N-API code that handles the formatting we encode all values into 64bit based representations (stored in the `data` property). We use a set of enumeration tags (stored in the `tags` property) to track what the kind of the corresponding 64bit value is. We have special handling for string and object property values, described below, that use the `strings` array and the `properties` map.

When implementing a sematic logging system the key invariant that needs to be preserved is that the values of each argument must not be changed between the time when the log call happens and when the argument is processed for formatting. Certain values including booleans, numbers, and strings, are immutable according to the language semantics so we can just copy the values/references directly into our in-memory array. However, in cases where the argument is a mutable object we must take explicit actions. A simple solution would be to JSON `stringify` the argument immediately, and while this prevents mutation and allows semantic formatting, it compromises possible performance gains we are looking for. Instead we recursively flatten the object into the in-memory buffer with the code below:

```
1  function addExpandedObject(obj, depth, length) {
2    //if the value is in a cycle
3    if(this.jsonCycleSet.has(obj)) {
4      this.addTagEntry(CycleTag);
5      return;
6    }
7
8    if(depth === 0) {
9      this.addTagEntry(DepthBoundTag);
10     return;
11   }
12
13   //Set processing as true for cycle detection
14   this.jsonCycleSet.add(obj);
15   this.addTagEntry(LParenTag);
16
17   let lengthRemain = length;
18   for (const p in obj) {
19     if (lengthRemain <= 0) {
20       this.addTagEntry(LengthBoundTag);
21       break;
22     }
23     lengthRemain--;
24
25     this.addPropertyEntry(p);
26     this.addGeneralEntry(obj[p], depth - 1, length);
27   }
28
29   //Set processing as false for cycle detection
30   this.jsonCycleSet.delete(obj);
31   this.addTagEntry(RParenTag);
32 }
```

This code is part of the transition from the formats and logger call arguments that is performed in the *Message processing* component shown in Figure 2. This code starts off by checking if we are either in a cycle or the depth bound has been reached. If either of these occur we put a special tag, `CycleTag` or `DepthBoundTag` respectively, into the tags array and return. If not we continue with the pre-order traversal of the object graph by updating the cycle info, adding the special `LParenTag` to the buffer, and enumerating the object properties. For each property we check if the length bound is met, adding the special `LengthBoundTag` and breaking if needed, if not we add the property and value information to the in-memory buffer. The property value `p` is a special string value which is very likely to be repeated so we use a map from properties to unique numeric identifiers

to compress and convert them into a number which can be stored in the `data` component of the in-memory buffer via the `addPropertyEntry` function. The value associated is recursively processed via the `addGeneralEntry` call which switches on the type of the value, booleans and numbers are converted directly to `Float64` representations, strings are mapped by index in the `strings` array, and `Date` objects are converted to 64bit timestamps using the `valueOf` method. Similar to standard JSON formatting other values are simplified but, instead of just dropping them, we use a special `OpaqueValueTag`. After all the properties have been processed we update the cycle tracking information and add a closing `RParenTag`.

To illustrate how this code works in practice consider the case of processing the object argument:

```
{
  f: 3,
  g: 'ok',
  o: { g: (x) => x }
}
```

The resulting state of the in-memory buffer is shown in Figure 5. In this example we can see how the object structure has been flattened into the buffer with the `'{'` and `'}'` tags denoting the bounds of each object. Each of the properties has been registered in the map (with a fresh numeric identifier) and this is stored in the `data` array. For example the property `g` is mapped to 1 and in both occurrences in the object structure the entry in the `tags` array is set to `'p'` for property and the value in the `data` array is set to match the corresponding numeric identifier 1. The numeric and boolean values are mapped in the obvious way, directly for the number and to 0/1 for the boolean with the appropriate tag values set. Similar to the properties the string value `'ok'` is mapped to an integer index, index 0, in the `strings` array. Finally, since functions are not serializable, the value associated with the `id` property is dropped and we simply store the `OpaqueValueTag` (`'?'`) in the corresponding position of the `tags` array.

Message Staging The in-memory design allows us to support *Design Principle 3* by buffering a window of high detail messages, in case they are needed for diagnostics, and then asynchronously filtering and flushing out high level telemetry messages to stable storage. Whenever a message is written into the in-memory buffer we do a simple `writeCount` check, to limit the flush rate, on the number of messages written since the last flush. If more than this threshold have been written we schedule an asynchronous flush process on the event-loop. The flush code filters the messages based on the current *emit* level and processes all messages that exceed the specified `age` or `memory` thresholds. The implementation of the filter/process step from Figure 2 is:

```
1  function processMsgs(pos, buff, dest, age, size) {
2    let now = new Date();
3    let cpos = pos;
```

```
tags: [ {, p, n, p, s, p, {, p, ?, }, } ]
data: [ _, 0, 3, 1, 0, 2, _, 1, _, _, _ ]

strings: [ 'ok' ]
props:   { f -> 0, g -> 1, o -> 2 }
```

Figure 5. In-Memory buffer processing example

```
4    while(cpos < buff.entryCount()) {
5      if(!ageCheck(buff, cpos, now, age) ||
6         !sizeCheck(buff, cpos, size)) {
7        return;
8      }
9
10     if(!isEmitLevelEnabled(buff.data[cpos])) {
11       cpos = scanAndDiscardMsg(buff, cpos);
12     }
13     else {
14       cpos = copyMsg(buff, cpos, dest);
15     }
16   }
17 }
```

This code works by looping over the messages in the in-memory buffer passed in, `buff`, and will process messages in the buffer until the `ageCheck` or `sizeCheck` conditions no longer hold. These conditions check if a message is older than the specified `age` measured from the current time and that overall memory consumption is less than `size` respectively. The intent behind the age limit is to prevent the overly eager discarding of log messages that might be relevant for debugging. While the intent behind the size condition is to ensure that an application which has a heavy burst of logging will not cause excessive memory consumption even if the messages have not aged out yet.

If an issue, currently defined as an unhandled exception or an exit with non-zero code, is encountered during execution the logger will immediately flush all the logger messages to the serializer regardless of age or level. This ensures that there is maximal detail for the developer when trying to diagnose the issue. LOG++ also supports programatic force-flushing of the log, with the `forceFlush` method, for cases where there is an issue that the developer wants to understand but does not result in a hard failure.

Emit Callback Once the in-memory buffer has been processed the logger passes the filtered set of log messages to the background *Async Formatter* (described in Section 3.3) which, when formatting is complete, will default to invoking a callback into the JavaScript application. For common cases we provide the simple default callback that writes to `stdout` or, if a `stream` is provided, to write the logs out through it. In the most general case a user can configure the logger with a custom callback, called with the formatted log data, which can perform any desired post processing and send the data to arbitrary (or multiple) sinks.

3.3 Native Formatting and Optional Native Emit

The background *Async Formatter* in Figure 2 is implemented in Native code using the N-API [18] module. This code is not part of the GC managed host JavaScript engine so we must fully copy any data needed before starting the background processing (a limitation we discuss relaxing in Section 3.5). However, once the data copy is done the formatting thread can run in parallel to the main JavaScript thread and can use an optimized C++ formatting implementation. This allows us to reduce the overall formatting cost and also to move this cost entirely off the hot-path of JavaScript execution.

In addition to background formatting Log++ also supports background emitting shown as the *Async Transport* block in Figure 2. Instead of moving formatted data back into the JavaScript engine to output we can directly write this logging data directly to a file, `stdout`, or a remote URL provided at configuration time from the native code layer. This allows us to skip re-marshaling the formatted data into a JavaScript string as well as reduces the number of JavaScript callbacks and associated time spent on the main thread there.

By default the native formatter produces JSON compliant output. However, this can be verbose to send and store so we offer serialization format and compression options for optimization. In addition to JSON a user can configure the logger to format to a simple binary encoding format which is not human readable but more compact and computationally efficient to encode into. With either format the logger can also be configured to compress the formatted data as well using the `zlib` library already included in Node.js core.

3.4 Logging APIs

As described in *Design Principle 4* there are frequent cases where a simple call to a log statement is insufficient and a developer would normally need to add explicit logging-specific logic. To avoid this clutter, and potential source of errors, we provide a richer set of logging primitives:

- *conditional logging* which allows a user to specify a condition that guards the execution of the log statement. This eliminates the need to introduce logging specific control flow into the code.
- *child loggers* which create a logger object that prefixes all log operations called on it with a specified value (used frequently for logging in a subcomponent).
- *request specific logging* which allows a developer to set a different logging level and enabled categories for a specific `RequestID`. This is a common need for cloud-based applications, particularly when participating in a sampling distributed logging system.
- *interval bounding* which writes a log message at the start of an interval and accepts a payload which can be accessed when writing the end event for the interval. This allows the transparent, instead of explicit, propagation of the relevant data between the log actions.

An example of the utility of these loggers, conditional logging, is in Figure 1 where the condition on line 14 was introduced solely for logging purposes. Using our conditional logging APIs this check and log combo can be replaced by a single line `logger.warnIf(!ok, "Error...")`.

3.5 Custom Runtime Implementation

The previous sections described the Log++ logger as it can be implemented as a purely userspace module in Node.js and as it is publicly available. However, there are issues, optimizations, and improvements that require close interaction with the runtime and/or compiler. This section describes these and how we have explored their implementation in Node-ChakraCore [20], the version of Node.js running the ChakraCore JavaScript engine from Microsoft.

Format and Mutability Checking Two programming errors that are specific to logging are mismatched arguments used for the provided format specifiers and accidental side-effects in logging related code. Log++ takes the view that logger calls should, if all possible, not cause an application failure and instead note invalid arguments in the output formatting. Previous work [28] has shown how to include static type checking for log arguments as well. A similar approach can be taken to combine work on purity analysis [5, 16] to ensure that any code executed in a log statement does not modify externally visible state. These features require at least support in a linter, such as *eslint* [8], or integration with the language itself.

Zero-Cost Disabled Logging Developers have low trust that logging disabled by setting levels or categories will not continue to impact performance. Thus, projects often experience frequent code churn as logging is added for a task and then removed afterwards to avoid performance issues. This wastes substantial developer time, creates opportunities to introduce accidental regressions, and prevents the development of a comprehensive base of logging code in an application.

To ensure that disabled log statements truely are zero-cost requires cooperation with the language spec and the JIT. To avoid parasitic costs of evaluating log arguments that will be immediately discarded, like line 8 in Figure 1, we must lazily evaluate them after the level, category, and (optional) condition checks have completed. We also want the JIT to be particularly aware of the guards around log statements, aggressively optimizing the guard paths, and performing dead-code-elimination or motion on computations that are only used for log arguments.

Fast Handling for Strings and Properties JavaScript object properties and strings are frequently occurring values in parts of a log message. If we are limited to a purely JavaScript and N-API interface we are forced to treat the properties as

strings and we must defensively copy the strings when processing them in the *emit processor* staging phase otherwise the JavaScript engine GC may move them while the formatter is accessing the underlying memory creating a data race and corruption.

Modern JavaScript engines use internal numeric *Property Identifiers* instead of strings when dealing with object properties. These are much more compact and efficient to process for both the engine and, in our logger, would allow us to copy a single integer into our in-memory buffer instead of creating our own indexing scheme. Thus, we expose a new JavaScript API, `loggerGetPropertyId`, which returns the internal numeric identifier for a property string to use in the message staging.

To avoid costs associated with data marshaling and copying string data in the formatter we need to introduce three APIs. The first is a pair of methods, `loggerRefString` in JavaScript which will tell the JavaScript engine that it needs to reference count, flatten, and intern a string (if needed) with a unique numeric value (which is returned), and a method `loggerReleaseString` which decrements the reference count and unpins the string if needed. These allow us to avoid copies when marshaling the string in the message staging phase. As various JavaScript engines use different internal widths for characters we add `getNativeStringCharSize` to determine if strings are `utf8` or `utf16` encoded. In combination with the methods `getRawBytes` to get the underlying buffer and `createStringWithBytes` to create a string this allows us to avoid any encoding conversions during the formatting.

Priority Aware I/O Management The final optimization we look at is to ensure that logging related I/O and computation do not interfere with high-priority code responding to user actions. In our implementation the message staging and uploading of formatted log data is done on the UV event loop. This loop does not have any notion of priority so we may accidentally block code responding to a user action with code that is processing log data. To avoid this we could add a special *log work queue* to the existing Node event loop processing algorithm but we believe that the notion of priority is fundamentally valuable and should be added to Node explicitly [14]. This *priority promise* construction makes it trivial to add logging related callbacks at a background task level to ensure that logging activities are completed in a timely manner without impacting user responsiveness.

4 Evaluation

Given the implementation of Log++ from Section 3 this section focuses on evaluating the resulting system and how it meets the design goals outlined in Section 2.

For the evaluation we use four microbenchmarks, listed below, which are each run for 10k iterations. We also use a server based on the popular *express* [9] framework which provides a *REST* API for querying data on S&P 500 companies. All of the benchmarks are run on an Intel Xeon E5-1650 CPU with 6 cores at 3.50GHz, 32GB of RAM, and SSD. The software stack is Windows 10 (17134) and Node v10.0.

Table 1. Timings for each logging framework on 10k iterations with given format. Speedup is the min-max speedup relative to the other logging frameworks.

Program	Basic	String	Compound	Compute
Console	883 ms	852 ms	1064 ms	1239 ms
Debug	202 ms	200 ms	282 ms	469 ms
Bunyan	477 ms	531 ms	603 ms	920 ms
Pino	188 ms	190 ms	296 ms	630 ms
Log++	89 ms	93 ms	155 ms	304 ms
Speedup	2.1-9.9×	2.0-9.2×	1.8-6.9×	2.1-4.1×

```
//Basic
log.info("hello world -- logger")

//String
log.info("hello %s", "world")

//Compound
log.info("hello %s %j %d", "world", { obj: true }, 4)

//Compute
log.info("hello at %j with %j %n -- %s",
         new Date(), ["i", { f: i, g: "#" + i }],
         i - 5, (i % 2 === 0 ? "ok" : "skip"))
```

4.1 Microbenchmarks

Our first evaluation is with current state of the art Node.js logging approaches. These include the builtin `console` methods, the `debug` [6] logger, the `bunyan` [3] logger, and the `pino` [23] logger. Each benchmark was run 10 times discarding the highest and lowest times and reporting the average of the remaining runs.

The results in Table 1 show the wide performance variation across logging frameworks (spanning nearly a factor of 10×). Across all benchmarks Log++ is consistently the fastest logger, by a factor of at least 1.8-2.1×, when compared to the best performing of the existing logging frameworks.

4.2 Logging Optimization Impacts

To understand how much each of our design choices and optimizations contributed to this performance we look at the performance impacts of specific features in Log++. Table 2 shows the Log++ baseline, the logger when we disable the background formatting thread (*Sync-Lazy*), the logger when we disable background formatting and disable batching of log messages in the in-memory buffer (*Sync-Strict*). We also look at how discarding before formatting and disabling log statements via multi-level logging features impacts performance. The *Levels (50%)* row shows the performance when 50% of the log statements are at the `INFO` level and 50% are

Table 2. Baseline performance in *Log++* row with disabled background format in *Sync-Lazy* and disabled background format & disabled lazy batch processing in *Sync-Strict*. The *Levels (50%)* and *Levels (33%)* rows show performance when 50% of log messages are in-memory level only and when 33% are in-memory only and 33% are entirely disabled respectively.

Program	Basic	String	Compound	Compute
Log++	89 ms	93 ms	155 ms	304 ms
Sync-Lazy	220 ms	216 ms	389 ms	636 ms
Sync-Strict	659 ms	788 ms	1035 ms	1323 ms
Levels (50%)	67 ms	72 ms	137 ms	223 ms
Levels (33%)	61 ms	65 ms	129 ms	189 ms

Table 3. Comparison of log messages using *expando* formats vs. manual computation and formatting of values for *hostname*, *application name*, *wall time*, and a monotonic *timestamp*

Program	Host	App	Wallclock	Timestamp
Explicit	8533 ms	65 ms	192 ms	41 ms
Expando	33 ms	32 ms	41 ms	36 ms

at the *DETAIL* level (processed into the in-memory buffer but discarded before format). The *Levels (33%)* row shows the performance when 33% of the log statements are at the `INFO` level, 33% are at the *DETAIL* level, and 33% are at the *DEBUG* level (entirely disabled).

The results in the *Sync-Lazy* and *Sync-Strict* rows from Table 2 show the impacts of the background formatting and the in-memory batching. Disabling the background formatter thread shows that the speedup by offloading formatting to the background is substantial (around 2× on the benchmarks). However, the impact of disabling the batching and lazy processing of the in-memory buffer is also significant. In our benchmarks we see as much as another 3.6× slowdown.

The use of *expando* macros in the formats, Section 3.2, can also play a large role in improving the performance of logging. Table 3 examines the performance difference when logging values using these macros vs. the cost of manually computing, formatting, and logging them.

As seen in Table 3, in cases such as adding the *hostname* of the current machine or the very common desire of including the current date/time (*wallclock*), there can be huge performance gains, 258× and 4.7× respectively, when using the expando macros. In other cases, such as the name of the current *app* or a monotonic *timestamp* value, the performance gains are a smaller but non-trivial 51% and 12%. Instead the benefit is primarily in simpler/cleaner logging code.

4.3 Logging Performance

The previous sections evaluated the performance of Log++ with respect to other loggers on core logging tasks and explored the impacts of various design choices using microbenchmarks. This section evaluates the impact of logging on a lightweight *REST* API service that supports querying data on S&P 500 companies. We use *autocannon* [2] in the default load generation setting to create a consistent load for a 10 second run on the service. For comparison we include the builtin `console` methods and the `pino` [23] logger in addition to Log++ in the default setting.

Table 4. Logging performance on the *REST* service server for `console.log`, `pino` and `Log++`. Also modified to take advantage of the `Log++` multi-level logging functionality in the *Log++ (levels)* row. Average and standard deviation values for response latencies are shown along with the average number of requests served per second.

Logger	Latency (avg)	Latency (stdev)	Req./s (avg)
Console	1.18 ms	0.83 ms	6668
Pino	0.89 ms	0.70 ms	8133
Log++	0.67 ms	0.80 ms	8645
Log++ (levels)	0.58 ms	0.77 ms	8958

This application highlights the tension between using logging as a telemetry source vs. a diagnostic tool. We updated it to use two logging levels, `DETAIL` and `INFO`. In the default runs we log at both levels and include a case, *levels*, where Log++ logs in-memory for the higher detail level but only emits at the lower level.

The results in Table 4 show that using a logging framework designed for modern development needs and built with performance in mind can have a substantial impact on an application. In terms of responses processed per second Log++ increases the server throughput by 30% from 6668 to 8645 requests per second. Further, Log++ decreases the response time by 43% from 1.18ms to 0.67ms. Despite using buffers and batched processing, which could in theory increase variability of the response latency, the standard deviation of the responses actually decreases slightly as well.

The results in Table 4 also show that, in addition to the improvements seen by using Log++ as a drop in replacement, it is possible to further improve the logger behavior by refactoring the logging statements to take advantage of the multi-level logging capabilities. For the *Log++ (levels)* row the application is changed to write log statements that are relevant for debugging, but not for general telemetry, at the `DETAIL` level. This results in their being stored in the in-memory buffer, if needed for diagnostics, but not formatted and emitted. As a result the throughput increases a further 4% to 8958 and the latency goes down an additional 13% to 0.58ms on average.

4.4 Logging Data Size

The final metric we evaluate is how Log++ can be used to reduce the amount of storage and network capacity consumed by logging data. Table 5 shows the log size generated per second when running the server benchmark with compression enabled (the *Compressed* column) as well as the impact of multi-level logging not needed to format/emit all log data.

Table 5. Log data generated per second on the *REST* service server for Log++ and modified to take advantage of the Log++ multi-level logging functionality in the *Log++ (levels)* row. Raw logging output size (*Raw* column) and log data size after deflate compression (*Compressed* column).

Logger	Raw	Compressed
Log++	2540 kb/s	137 kb/s
Log++ (levels)	1176 kb/s	84 kb/s

As shown in Table 5 both compression and the ability to discard detailed (and noisy) messages in the multi-level setup provide large reductions in the data sizes that need to be transmitted and stored. As expected compression is very effective on log data, reducing the size by 94.5% from 2.54 MB/s to 0.13 MB/s. The ability to discard noisy messages, once it is determined they are not interesting for debugging, also has a large individual impact and reduces the data size by 53.7%. Combined these two optimizations result in a total data size reduction of a massive 96.7% going from 2.54 MB/s to just 0.08 MB/s.

5 Related Work

While logging is a fundamental part of many software development workflows it has received relatively little attention overall from the academic community and, to the best of our knowledge, there is no prior work explicitly on the design of core logging frameworks.

Logging State of the Art: Existing logging frameworks provide simplified versions of some of the systems described in this work. Recently the concept of semantic logging has appeared in loggers for Java [13] and C# [24]. However, the prevalence of logging JSON style objects in JavaScript, v.s., mostly primitive values in Java or C#, presents a challenge that we resolve efficiently with the flattening algorithm in Section 3.2. Buffering and formatted logging are also a very common design choices, e.g., in pino [23] or bunyan [3], but they focus on buffering formatted data or using pure JSON for the structuring. In contrast this work buffers compound data + message format information in our in-memory buffer design and allows both JSON style formats as well as parsable `printf` style messages.

Logging Practices: The closest theme of prior research is focused on empirical investigation of logging use in practice and tools to support good logging practices [11, 32]. Using a large scale evaluation of OSS projects [32] studied code changes involving logging code to understand how and why developers were using logging. Work on closed source applications [11] reached many of the same conclusions. These studies provided valuable insights which were used in distilling the design principles used in this work.

Improved Logging: A larger area of work has been into techniques to support best practices for logger use. From a type system perspective [28] developed a type system and checker to ensure format specifiers and their arguments were well typed. Work in LCAnalyzer [4] proposes techniques to help developers with finding poor logging uses. Other work develops tools, such as LogAdvisor [34], LogEnhancer [33], and ErrLog [31], which help developers identify locations and values which should be logged to support later diagnostics or analysis operations.

Log Analysis: Work on the topic of using logs to support other software development activities is more extensive. This work includes post mortem debugging [15, 22, 29], anomaly detection [10], feature use studies [12], and automated analysis of performance issue root causes [17]. This body of work highlights the potential value of high quality logging data and the opportunities for research and tooling that depends on it.

6 Conclusion

This paper introduced a set of design principles for logging with the view that logging is a fundamental part of the software development and deployment lifecycle. By thinking of logging this way and how it can be closely coupled with the rest of the language and runtime for best performance and usability we developed a novel logging system with several innovative features. As a result Log++ outperforms existing state of the art logging frameworks and represents an important development in advancing the state of the art for modern modern cloud, mobile, and IoT development workflows.

Acknowledgments

I would like to thank Matteo Collina, Matthew C. Loring, and Mike Kaufman for their insights into logging and useful feedback on this work. Thanks to Arunesh Chandra for his help with N-API and thoughts on how to use it effectively. I would also like to thank the anonymous reviews for their excellent feedback and help in improving this work.

References

[1] AppInsights 2018. AppInsights. https://azure.microsoft.com/en-us/services/application-insights/.
[2] autocannon 2018. autocannon. https://github.com/mcollina/autocannon/.
[3] bunyan 2018. bunyan. https://github.com/trentm/node-bunyan/.
[4] Boyuan Chen and Zhen Ming Jiang. 2017. Characterizing and Detecting Anti-Patterns in the Logging Code. In *ICSE*.
[5] Jong-Deok Choi, Michael Burke, and Paul Carini. 1993. Efficient Flow-sensitive Interprocedural Computation of Pointer-induced Aliases and Side Effects. In *POPL*.
[6] debug 2018. debug. https://github.com/visionmedia/debug/.
[7] elastic 2018. Elastic Stack. https://www.elastic.co/products/.
[8] eslint 2018. eslint. https://eslint.org/.
[9] express 2018. express. https://expressjs.com/.

[10] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *ICDM*.
[11] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where Do Developers Log? An Empirical Study on Logging Practices in Industry. In *ICSE*.
[12] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. 2012. The Unified Logging Infrastructure for Data Analytics at Twitter. *Proceedings VLDB Endowment* (2012).
[13] log4j 2018. Log4j. https://logging.apache.org/log4j/2.x/.
[14] Matthew C. Loring, Mark Marron, and Daan Leijen. 2017. Semantics of Asynchronous JavaScript. In *DLS*.
[15] Leonardo Mariani and Fabrizio Pastore. 2008. Automated Identification of Failure Causes in System Logs. In *ISSRE*.
[16] Mark Marron, Darko Stefanovic, Deepak Kapur, and Manuel Hermenegildo. 2008. Identification of Heap–Carried Data Dependence Via Explicit Store Heap Models. In *LCPC*.
[17] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *NSDI*.
[18] NAPI 2018. N-API. https://nodejs.org/api/n-api.html.
[19] Node 2018. Node.js. https://nodejs.org/.
[20] NodeChakraCore 2018. Node with ChakraCore. https://github.com/nodejs/node-chakracore/.
[21] NSolid 2018. N|Solid. https://nodesource.com/products/nsolid/.
[22] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and Challenges in Log Analysis. *Communications ACM* (2012).
[23] pino 2018. pino. https://github.com/pinojs/pino/.
[24] Serilog 2018. Serilog. https://serilog.net/.
[25] Weiyi Shang, Zhen Ming Jiang, Bram Adams, Ahmed E. Hassan, Michael W. Godfrey, Mohamed Nasser, and Parminder Flora. 2011. An Exploratory Study of the Evolution of Communicated Information about the Execution of Large Software Systems. In *WCRE*.
[26] Splunk 2018. Splunk. https://www.splunk.com/.
[27] StackDriver 2018. Stackdriver. https://cloud.google.com/stackdriver/.
[28] Konstantin Weitz, Gene Kim, Siwakorn Srisakaokul, and Michael D. Ernst. 2014. A Type System for Format Strings. In *ISSTA*.
[29] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. 2010. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *ASPLOS*.
[30] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *USENIX*.
[31] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *OSDI*.
[32] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing Logging Practices in Open-source Software. In *ICSE*.
[33] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. 2011. Improving Software Diagnosability via Log Enhancement. In *ASPLOS*.
[34] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R. Lyu, and Dongmei Zhang. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. In *ICSE*.