**Asynchronous Logging**

Asynchrnous and distributed logging solutions

Badal

**Abstract**

This document describes and compares different approaches for application logging for better application performance and log files readability and maintainability.

Table of Contents

Problem Statement 3

Functional Requirement 3

Non-Functional Requirement 4

Performance 4

Scalability 4

Performance Benchmark 4

Replication Environment & Testing 4

API Guideline 5

Logging Solutions 5

Application Logging (Application Level) 5

Asynchronous PHP logging 5

Asynchronous Log4j2 with Disruptor 6

Distributed Logging 8

HTTP Post 8

Message Queue 9

NIO/Netty 10

Message Queue (Aeron) 11

Distributed Logging with Apache Bookkeeper 12

Comparison and Benchmarking 13

Log Analyzer 13

Code Repository 14

# Problem Statement

Logging is a fundamental part of applications. Every application has a varying flavor of logging mechanism. A well-designed logging system is a huge utility for system administrators and developers, especially the support team. Logs save many valuable hours for both the support team and developers. As users execute programs at the front end, the system invisibly builds a vault of event information (log entries) for system administrators and the support team.

Logging statements should have no or little impact on the application’s behavior specially slowing the application down.

Increasing traffic increases load on our application server, increasing amount of time spent for writing application logs. As the log file grows in size every IO operation become even more costly.

So maintaining a concise logging mechanism is essential for system administrators, developers and the support team along with application performance.

To address the above requirement develop an asynchronous logging framework with high throughput, suitable for high load so that logging operation does not block request processing and thus application performance.

# Functional Requirement

1. Write application log asynchronously. Writing logs asynchronously ensures that application performance is not compromised.
2. Framework must have well defined APIs for logging. This APIs must accept log file name, log level, logging system details, timestamp and a log message to ensure that log file can be helpful for debugging.
3. The framework must enforce formatting to form a well defined and structured logging for analyzers to analyze the log easily.
4. Logging framework must use daily rolling file appender splitting and creating new file everyday for better maintenance and readability or a client specified file name handing over the responsibility to client to manage log file names.
5. The framework must support & maintain multiple log files simultaneously.
6. Framework must have the optional capability of splitting the log file when a predefined max size is reached.
7. The framework may write log at a centralized location or at individual nodes. A centralized location will definitely give better maintenance.
8. The framework should have a fallback mechanism in case the logging framework fails to log at desired location
9. The sequence of logs should remain in tact.

# Non-Functional Requirement

The framework should work across platform with following non-functional features

## Performance

Logging must not impact the application performance. The logging framework must ensure high performance. The framework must support 200,000 logging request per second to 300,000 requests per second from multiple clients.

The framework must support high through put approximately 10-15% of the incoming texts to be logged could be in MBs.

## Scalability

The framework should be highly scalable to support at least 400,000 requests per second from multiple sources.

# Performance Benchmark

Do a performance benchmark (***messages/sec*** higher the better) with all major logging framework available like log4j, logback etc.

# Replication Environment & Testing

*Setup a testing environment that can handle the desired no of request per second (~ 300,000 request /second)*

*Test Scenario: Use any tool to create desired no of threads to send required concurrent requests. The test environment will be able to handle the request.*

*Verification 1: Verify the accuracy and readability of the logs generated. There will not be any message loss, logs should be well structured and any log analyzer can easily parse it.*

*Verification 2: Verify message formatting and server, thread and timestamp information of written log. Server information will help us grouping logs based on server similarly thread and timestamp will also help us sequencing the events occurred in the application.*

*Verification 3: Verify the daily rolling file appender behavior of the generated log or if file names are correct as per the incoming log requests and messages are logged properly in different files.*

*Verification 4: Verify optional split file behavior when predefined file size is reached behavior.*

*Verification 5: Do a performance benchmark as described in performance benchmark section.*

*Verification6: Verify the solution work in all the platforms like Java, PHP etc.*

# API Guideline

The logging messages should include host server, thread id, formatted timestamp, logging source class/file name and the actual log message. Logically it should include the formatting and structure like log4j or logback.

# Logging Solutions

There are multiple solutions to the problem. For a simple application with moderate load application logging with some asynchronous logging frameworks could be sufficient, for applications with very high load a distributed logging solution is desirable.

## Application Logging (Application Level)

This is not distributed logging, in this approach application logs are generated on the same server on which application is deployed. Platform specific asynchronous logging framework can be used for this type of logging.

### Asynchronous PHP logging

#### Architecture

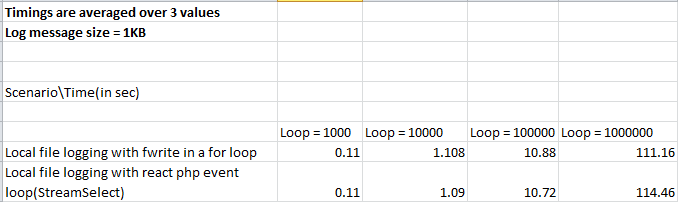
The logging in server application is normally done using a blocking call like *fwrite*. As the load on the application grows, the logging call will result in lower throughput. We need to make the call to logging function non-blocking. Following are the ways to achieve non-blocking I/O from PHP

1) Use libraries which provides asynchronous I/O like react PHP

2) Write a PHP extension in some other language like C/C++/Javascript to achieve non-blocking I/O

**React PHP**

* React PHP uses event loop to achieve non-blocking I/O with its event loop, stream and socket classes.
* Problems: The stream class maintains its own buffer which can be written on to asynchronously and when the event loop is eventually run, it will flush the data to file/socket etc. The problem is that ‘run’ method of the loop blocks the code. So eventually we get more or less the same performance with this approach



**PHP extension**

* Started on a POC with C++
* Javascript: TBD

### Asynchronous Log4j2 with Disruptor

Asynchronous logging can improve application performance by executing the I/O operations in a separate thread. Log4j 2 makes a number of improvements in this area using Asynchronous Loggers, LMAX Disruptor technology and Asynchronous Appenders.

#### Architecture

Thread

Logging

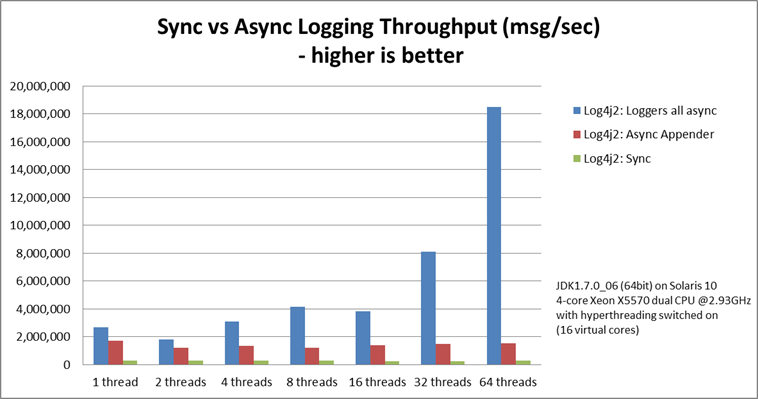
#### Advantages

* Higher throughput. With an asynchronous logger application can log messages at 6 - 68 times the rate of a synchronous logger.
* Lower logging latency. Latency is the time it takes for a call to Logger.log to return. Asynchronous Loggers have consistently lower latency than synchronous loggers or even queue-based asynchronous appenders.
* Prevent or dampen latency spikes during bursts of events. If the queue size is configured large enough to handle spikes, asynchronous logging will help prevent your application from falling behind (as much) during sudden bursts of activity.

#### Disadvantages

* Error handling. If a problem happens during the logging process and an exception is thrown, it is less easy for an asynchronous logger or appender to signal this problem to the application.
* If logging is part of our business logic, for example if you are using Log4j as an audit-logging framework, better to synchronously log those audit messages.
* MapMessage and StructuredDataMessage are mutable by design: fields can be added to these messages after the message object was created. These messages should not be modified after they are logged with asynchronous loggers or asynchronous appenders; we may or may not see the modifications in the resulting log output.

#### Performance



Blogpost[<https://www.grobmeier.de/log4j-2-performance-close-to-insane-20072013.html>] and [ https://logging.apache.org/log4j/2.x/manual/async.html] has all the performance benchmarking for throughput and latency with increasing no of threads.

#### Recommendation

Asynchronous Loggers internally use the Disruptor, a lock-free inter-thread communication library, instead of queues, resulting in higher throughput and lower latency. So if we are working on a small to large scale application with moderate load on the server we can definitely use asynchronous log4j2 with disruptor, however log files in this approach will be at different application servers so a log analyzer after collecting all the log files from different nodes could be used for better log analysis.

## Distributed Logging

In distributed logging, distributed applications sends messages to log to a centralized application that manages writing messages to log files. Managing the log files, message ordering, centralized log application scalability, fault tolerance are the major factors while deciding an optimal solution for distributed logging.

### HTTP Post

In this approach a REST endpoint is exposed for the client to send JSON messages to a hosted application, which internally uses asynchronous log4j2 with disruptor to log messages to a cartelized location. Message ordering, hosted application scalability, and varying message length are the matter of consideration in this approach.

#### Architecture

App Server

Client Application

Request

Async Logging Module

Async Logger

REST End point

Application Logic

Response

Load Balancer

Log

#### Advantages

* Easy to implement, server side application can be developed easily using log4j2 with disruptor or any asynchronous logging framework with high throughput or NIO based solution could be used.
* Any client can consume REST API easily.

#### Disadvantages

* Client side message ordering is a problem. Asynchronous logger will write log message, which could include timestamp but that timestamp could not be maintained while writing to log files. A log analyzer is needed to rearrange them while reading.
* The client application platform must support fire and forget code execution.
* Load balancer could be single point of failure so 100% fault tolerance may not be achieved.
* There will be a limitation with number of concurrent users that could be supported by the server side application, so it’s not highly scalable.

#### Performance

The application will be deployed in tomcat or other container so it depends on maximum parallel threads supported by the container.

#### Recommendation

This approach could be used when network latency (for lengthy messages to log), written log file ordering and scalability are not major concern. So for a moderate loaded application this could be really simple to implement and use.

### Message Queue

A low latency, high throughput, fault tolerant message queue could be use to solve distributed logging problem. We will be using Kafka for our case study. The client application will publish every log message to the message queue using a producer asynchronously. The consumer will consume those messages and write log files using log4j2 asynchronous logger with disruptor.

#### Architecture

Client Application

Request

Async Message Producer

Application Logic

Kafka

Response

App Server

Message Consumer

Async Logger

Log

#### Advantages

* A low latency, high throughput and fault tolerant message queue like Kafka will address scalability and fault tolerance.
* Message ordering in the queue will be maintained.

#### Disadvantages

* Kafka only provides a total order over messages within a partition, not between different partitions in a topic so message order will not be maintained based on client.
* Message length beyond certain limit (10KB for kafka) could be a problem in this approach.

#### Performance

Benchmark ://TODO

#### Recommendation

* Overall Kafka message queue can handle scalability, fault tolerance and within partition message ordering nicely. Lengthy message could be handled in any of the following ways -
* The best way to send large messages is not to send them at all. If shared storage is available (NAS, HDFS, S3), placing large files on the shared storage and using Kafka just to send a message about the file’s location is a much better use of Kafka.
* The second best way to send large messages is to slice and dice them. Use the producing client to split the message into small 10K portions, use partition key to make sure all the portions will be sent to the same Kafka partition (so the order will be preserved) and have the consuming client sew them back up into a large message.
* Kafka producer can be used to compress messages. If the original message is XML, there’s a good chance that the compressed message will not be very large at all. Use compression. codec and compressed.topics configuration parameters in the producer to enable compression. Zip and Snappy are both supported.
* Client message ordering can be handled using a log analyzer stack like ELK (Elasticsearch, Logstash & Kibana)

### NIO/Netty

Use Netty’s NIO client server framework, which greatly simplifies and streamlines network programming such as TCP and UDP socket server

#### Architecture

Channel Buffer

Channel Pipeline

Server Handler

SSLEncoder

StringEncoder

HttpRequestEncoder

Logs

Log4j2 Logger

#### Advantages

* Better throughput, lower latency
* Less resource consumption
* Minimized unnecessary memory copy

#### Disadvantages

* Scaling & fault tolerance problem
* Platform dependency (Java, C++)

#### Performance

*Result "benchmarkAsyncLog":*

*189.378 ops/s*

*# Run complete. Total time: 00:00:02*

*Benchmark Mode Cnt Score Error Units*

*NettyBenchmark.benchmarkAsyncLog thrpt 189.378 ops/s*

#### Recommendation

A good reason to use Netty is to improve the reliability of the connections and leave you to code what the connection does rather than worry about the details of everything, which can go wrong.

Netty may help you scale over 1K connections. However if you don't need so many connections you might find that simple code performs best.

### Message Queue (Aeron)

Aeron is a high-performance messaging system written in Java built with mechanical sympathy in mind, and can run over UDP, Infiniband or Shared Memory, using lock-free and wait-free structures.

#### Architecture

Client Application

Request

Async Message Producer

Application Logic

Aeron

Response

App Server

Message Consumer

Async Logger

Log

#### Advantages

* Efficient reliable UDP unicast, UDP multicast, and IPC message transport.

#### Disadvantages

* Scaling & fault tolerance problem
* Drivers are available for Java and CPP platform as of now.

#### Performance

*Benchmark (burstLength) Cnt Score Error Units*

*test1Producer 1 749123 666.218 ± 125.096 ns/op*

*test1Producer 100 596238 7496.482 ± 593.774 ns/op*

*test2Producers 1 1336303 980.003 ± 168.752 ns/op*

*test2Producers 100 1247303 12561.019 ± 544.367 ns/op*

*test3Producers 1 2197803 2514.564 ± 347.449 ns/op*

*test3Producers 100 1662335 28856.265 ± 1174.739 ns/op*

#### Recommendation

If there is a need for low latency transport protocols like capital markets use cases Aeron could be suitable as TCP is not suitable for low latency due to the tradeoff between bandwidth usage after loss and responsiveness.

### Distributed Logging with Apache Bookkeeper

BookKeeper is a highly scalable, reliable, replicated log storage based on commodity hardware. BookKeeper abstracts out certain complexities, such as replication, failure recovery, and consistency, while building web scale applications by providing simple constructs to store and retrieve sequential log entries

#### Architecture

Ledger Indexes

Ledger Index

Read

JBOD

Entry Log

JBOD

Write

SSD

Journal

#### Advantages

* Horizontal Scalability - BookKeeper scales in both IO and storage capacity seamlessly with addition of new commodity servers, thus enabling incremental elastic growth
* High Throughput - BookKeeper architecture for reads and writes is optimized for sequential disk IOs which leads to higher throughput on a single server
* Consistency - In case of failures and errors common in large scale distributed systems, BookKeeper provides a consistent view of the log entries to all readers

#### Disadvantages

* No disadvantages as such only complexity is cluster management using zookeeper

#### Performance

Following is the performance benchmark for 10 local bookies with 8 GB RAM systems -

*2016-08-17 12:00:44,901 - WARN - [main:BenchThroughputLatency@294] - (Parameters received) running time: 60, entry size: 1024, ensemble size: 3, quorum size: 2, throttle: 10000, number of ledgers: 1, zk servers: localhost:2181, latency file: latencyDump.dat*

*Data Size: 1024*

*2016-08-17 12:01:51,464 - INFO - [Thread-4:BenchThroughputLatency@202] - Finished processing in ms: 60007 tp = 10000*

*2016-08-17 12:01:51,464 - INFO - [main:BenchThroughputLatency@366] - Calculating percentiles*

*2016-08-17 12:01:51,700 - INFO - [main:BenchThroughputLatency@387] - 600097 completions in 60007 milliseconds: 10000 ops/sec*

*2016-08-17 12:01:51,985 - INFO - [main:BenchThroughputLatency@406] - 99th percentile latency: 6.877530123671932*

*2016-08-17 12:01:51,990 - INFO - [main:BenchThroughputLatency@407] - 95th percentile latency: 4.219469339766213*

*2016-08-17 12:12:38,832 - WARN - [main:BenchThroughputLatency@294] - (Parameters received) running time: 60, entry size: 10240, ensemble size: 3, quorum size: 2, throttle: 10000, number of ledgers: 1, zk servers: localhost:2181, latency file: latencyDump.dat*

*Data Size: 10240*

*2016-08-17 12:13:48,499 - INFO - [Thread-4:BenchThroughputLatency@202] - Finished processing in ms: 60008 tp = 10000*

*2016-08-17 12:13:48,500 - INFO - [main:BenchThroughputLatency@366] - Calculating percentiles*

*2016-08-17 12:13:48,676 - INFO - [main:BenchThroughputLatency@387] - 600101 completions in 60008 milliseconds: 10000 ops/sec*

*2016-08-17 12:13:48,938 - INFO - [main:BenchThroughputLatency@406] - 99th percentile latency: 170.01011817730884*

*2016-08-17 12:13:48,944 - INFO - [main:BenchThroughputLatency@407] - 95th percentile latency: 132.7319593693174*

*2016-08-17 12:22:47,960 - WARN - [main:BenchThroughputLatency@294] - (Parameters received) running time: 60, entry size: 102400, ensemble size: 3, quorum size: 2, throttle: 10000, number of ledgers: 1, zk servers: localhost:2181, latency file: latencyDump.dat*

*Data Size: 102400*

*2016-08-17 12:26:08,838 - INFO - [Thread-4:BenchThroughputLatency@202] - Finished processing in ms: 71205 tp = 307*

*2016-08-17 12:26:08,838 - INFO - [main:BenchThroughputLatency@366] - Calculating percentiles*

*2016-08-17 12:26:08,913 - INFO - [main:BenchThroughputLatency@387] - 21868 completions in 71205 milliseconds: 307 ops/sec*

*2016-08-17 12:26:08,956 - INFO - [main:BenchThroughputLatency@406] - 99th percentile latency: 30092.136051448702*

*2016-08-17 12:26:08,957 - INFO - [main:BenchThroughputLatency@407] - 95th percentile latency: 28950.243770318044*

#### Recommendation

BookKeeper is a highly scalable, reliable, replicated log storage based on commodity hardware. BookKeeper abstracts out certain complexities, such as replication, failure recovery, and consistency, while building web scale applications by providing simple constructs to store and retrieve sequential log entries.

BookKeeper provides capabilities like -

Horizontal Scalability - BookKeeper scales in both IO and storage capacity seamlessly with addition of new commodity servers, thus enabling incremental elastic growth.

High Throughput - BookKeeper architecture for reads and writes is optimized for sequential disk IOs, which leads to higher throughput on a single server.

Consistency - In case of failures and errors common in large scale distributed systems, BookKeeper provides a consistent view of the log entries to all readers.

# Comparison and Benchmarking

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Serial No | Solution | Scalability | Fault Tolerance | Message Ordering | Message Length | Comments |
| 1 | Platform Specific Async Client | Scalable | NA | NA |  |  |
| 2 | Log4j2 with Disruptor | NA | NA | NA | Increasing message length reduces performance |  |
| 3 | HTTP Post | Scalable with limitations | Yes, can be made fault tolerant with scaling | Client sent message order | Increasing message length reduces performance |  |
| 4 | Kafka Queue | Highly Scalable | Kafka cluster provides fault tolerance | Client sent message order | Increasing message length reduces performance. We can Partition the message |  |
| 5 | Netty | NA | No | Client sent message order | Increasing message length reduces performance |  |
| 6 | Aeron | Don’t support cluster till now | No | Client sent message order | Increasing message length reduces performance |  |
| 7 | Bookkeeper | BookKeeper is a replicated log service so highly scalable | BookKeeper is a replicated log service, it works on cluster so its fault tolerant | Client sent message order | Increasing message length reduces performance |  |

# Log Analyzer

Logstash is an open source tool for collecting, parsing, and storing logs for future use. Kibana is a web interface that can be used to search and view the logs that Logstash has indexed. Both of these tools are based on Elasticsearch, which is used for storing logs.

**ELK** setup has four main components

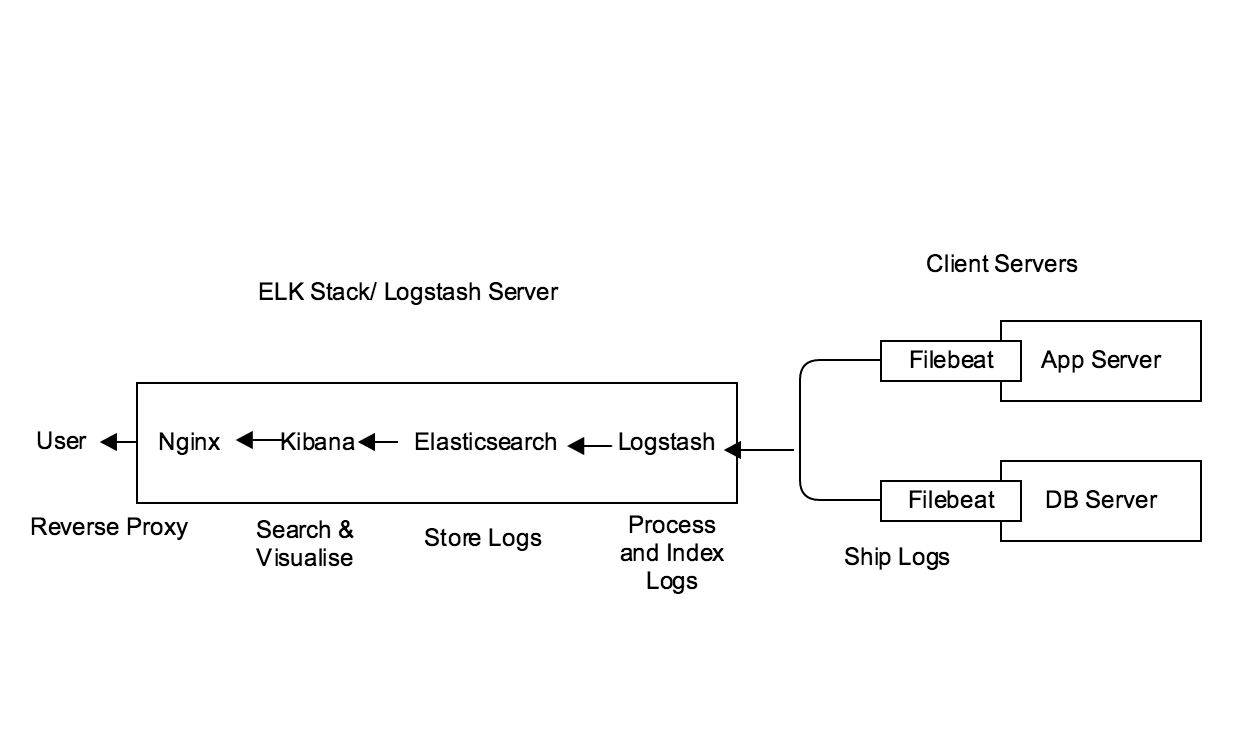
**Logstash**: The server component of Logstash that processes incoming logs

**Elasticsearch**: Stores all of the logs

**Kibana**: Web interface for searching and visualizing logs, which will be proxied through Nginx

**Filebeat**: Installed on client servers that will send their logs to Logstash, Filebeat serves as a log shipping agent that utilizes the lumberjack networking protocol to communicate with Logstash

The complete architecture looks like below -



We can follow any standard tutorial for ELK setup with Filebeat. Filebeat to Logstash communication may be over https, which requires SSL certificate.

Now that our logs are centralized via Elasticsearch and Logstash, and we are able to visualize them with Kibana, we should be off to a good start with centralizing all of your important logs. Remember that we can send pretty much any type of log or indexed data to Logstash, but the data becomes even more useful if it is parsed and structured with grok.

# Code Repository

Server side code can be found at *[https://github.com/badalb/Application-Log.git* ]