Advanced Logging and Forwarding Proposal

Improve existing logging package for customers and developers.

Objectives

- Allow customers to send log records to multiple destinations
    - file
    - syslog
    - TCP socket
- Allow customers to send log records to different destinations based on log levels
- Allow developers to specify discrete log levels as well as the standard trace, debug, info, ... panic
- Allow developers to keep existing code and logging API usage unchanged
- Emit log records asynchronously to reduce latency to the caller
- Support hot-reload of logger config

Current logging package

Currently the Mattermost server outputs logs to console and/or file, using plain text formatting or JSON. No customization of fields output or ordering is provided. Only one file can be specified and only one stream (console via stdout).

There is limited flexibility with respect to filtering which log records get emitted. A single log level can be assigned to file output and another to console. If a developer wants some debug output their only option is to enable “debug” or “trace” and deal with the fire-hose of output.

Customers wishing to use log aggregators such as Splunk, Sumo Logic, Loggly, Fluentd, etc..., must set up log forwarders or other mechanisms to get the logs into the aggregator. This must be done for each node in a Mattermost cluster.

Currently all log records must be fully written to file and/or console before the calling thread can resume work.

Multiple targets

Allow any combination of local file, syslog and TCP socket targets. Syslog and TCP can be configured with or without TLS. These three targets have been chosen to support the vast majority of log aggregators and other log analysis tools without having to install additional software.

The logging package will be capable of outputting to multiple instances of each target. For example, a configuration may specify that all log records at log level “error” output to a specific file while “info”, “debug”, and “trace” end up in a different file.

- File target supports rotation and compression triggered by size and/or duration
- Syslog target supports local and remote syslog servers, without or without TLS transport
- TCP socket target can be configured with an IP address or domain name, port, and optional TLS encryption

Discrete log levels

Current log levels are range based, where a log level specifies all integers between itself and zero. This will be extended to allow discrete log levels using any integer to MAX_INT.

Any log levels below 10 will continue to behave as ranges to zero, while 10 and above will be discrete. Targets can be configured with any number of log levels.

Applications using the logging package are encouraged to create a constants file where developers can register their log levels and avoid collision.

Developers can feel free to add specialized debugging output, generally turned off but available when needed, with just the overhead of a map lookup for most cases.

For example, someone working on websockets can create log entries for all aspects of websocket life-cycle using one or more discrete log levels. They can then configure a log target that only includes those websocket log levels to get a clean view of websocket activity that can be easily turned on/off.

Asynchronous logging

Logging to local file and console has predictable latency for low volumes of records. Higher volumes or remote targets add variability to write latency which often manifest as bursts of writes with intermittent pauses. We don’t want calling threads to wait during these pauses when they should be fulfilling user requests.

Logging to in-memory buffers will allow calling threads to return sooner and allow background threads to manage the variable latency of targets.

Two levels of buffer will be used. A top level buffer which accepts a partially formed log record and allows the caller to continue. Log records are only generated if at least one target has a matching log level. The results of checking for any targets matching a log level will be cached in a map of int->bool which is cleared any time the logging configuration changes.

When logging structs and other mutable structures, it is important to capture the information to be logged before it can be modified. This can be done via defensive copying or by serializing via the calling thread. Defensive copying requires writing clone methods for all structs plus the serialization code, and also incurs the copy overhead. Serializing to an intermediate format has the benefit of only requiring serialization code.

When model structs are eventually made immutable, this intermediate formatting can be done in the background.

Final formatting will be done in the background. Targets will be configured for JSON or plain text, and allow customization of which fields are output, in which order, and timestamp format can be specified. This will facilitate easier parsing for log aggregators.

In cases where the queue does become full, the logger can be configured to drop records and alert, or block until space is available (with timeout) which behaves similarly to the current logger.

Hooks are available to monitor queue sizes versus capacity.

Implementation

Asynchronous logging with multi-target support was added to the Mattermost server via the auditing system for version 5.22. This capability will be leveraged for general logging. Only the “mlog” package will change, and the existing APIs will remain.

It is to be decided how much of this functionality will be available in TE versus EE.

For initial release, there may only be UI to configure one of each target type.

Some performance considerations when choosing a logging engine:

- How many allocations are made for a logging line that does not emit a log record due to log level mismatch?
- How many allocations are made to fully serialize and format a log record? Does the JSON encoder used support zero allocation serialization?

- Is the logger suitable for logging to variable latency destinations? Does it support asynchronous logging?

Currently Mattermost server utilizes two logging engines, Logrus and Zap.

Logrus is arguably the most popular logger in the Go ecosystem, which is unfortunate as it has some of the worst benchmarks for memory and latency. Logrus' strength is in the large number of plugin targets (hooks) available.

Zap has very good benchmarks for memory and latency, however it is limited in where logs can be written (local file) and their formats (JSON).

The logging engine added in v5.22 for auditing is designed to provide the best of both of those loggers. Zero allocation JSON serialization, low allocation log record creation, low latency formatting, while providing a decent range of built-in targets and formatters. It also provides an adapter which can be used with any Logrus hook, thus making any Logrus hook work asynchronously.

Levels

```
{
        "ID": int,                  // unique id; register new ones in mlog/levels.go
        "Name": string,             // descriptive name
        "Stacktrace": bool          // true if stack trace should be generated
}
```

Targets

Log targets are defined by a common struct, with Options specific to the target type.

Log Target

```
{
        "Type": string,             // one of: "console", "file", "syslog", "tcp"
        "Format": string,           // "json" or "plain"
        "Levels": [],               // array of Levels
        "Options": {},              // map of options (see below)
        MaxQueueSize: int,          // Defaults to 1000
}
```

Console

```
{
```

```
        “Options”: {
                “Out”: string           // “stdout” or “stderr”
        }
}
```

File

```
{
        “Options”: {
                “Filename”: string,     // path and filename for logs
                “MaxAgeDays”: int,      // number of days until a rotation is triggered;
                                        // 0 means don’t rotate based on age
                “MaxSizeMB”: int,       // maximum file size before a rotation is triggered;
                                        // 0 means don’t rotate based on age
                “MaxBackups”: int,      // Maximum number of rotated files to keep;
                                        // oldest will be deleted; zero means don’t keep
                                        // rotated files
                “Compress”: bool        // true if files are compressed after rotation
        }
}
```

SysLog

```
{
        “Options”: {
                “IP”: string,           // IP address or domain of syslog server
                “Port”: int,            // Listening port of syslog server
                “Tag”: string,          // Typically the program name, machine name,
                                        // or node name
                “TLS”: bool,            // if true then connect via TLS
                “Cert”: string,         // filename of client certificate or base64 encoded cert
                                        // for TLS connections
                “Insecure”: bool        // If true then certificate check is not performed;
                                        // for testing only
}
```

TCP

```
{
        “Options”: {
                “IP”: string,           // IP address or domain of socket server
                “Port”: int,            // Listening port of socket server
                “TLS”: bool,            // if true then connect via TLS
                “Cert”: string,         // filename of client certificate or base64 encoded cert
```

```
                                    // for TLS connections
                "Insecure": bool    // If true then certificate check is not performed;
                                    // for testing only
}
```