flow.log message ID/filtering feature design sketch

Before writing this up, I ran this by Eddy Chan with sound waves for about an hour. We are in agreement generally.

Later, Philip Lisiecki also messaged me to say he's into it.

Introduction/motivation

A **message ID** is an integer present directly at a log call site (e.g., FLOW_LOG_INFO() macro for Flow) in logging code, distinct from all other log call sites' IDs in that module. (A **module** in Flow.log context is, basically, a library or application; for example Flow itself is a module; <some daemon> main code is a module; any Flow-logging library is a module. Since modules can be combined in various combinations depending on the application, their message IDs in the code must necessarily be able to clash. Any clash between message ID sets of different modules can be resolved by identifying the component that is logging that message – also present in the log line.) The message ID will typically then be logged in the output (usually log file; sometimes console; etc.) either in lieu of or in addition to the originating file, line, function (from C macros __FILE__, __LINE__, __FUNCTION__).

A message ID is not directly generated by the coder while coding. It is done automatically, at least once per release but potentially more if desired: a script scans all log call sites in the code; for any site with no ID (e.g., with ID=0) the log call site code is changed by the script by generating a new, distinct ID, which is equal to the largest ID so far, plus 1. (The max ID is, perhaps, saved somewhere in a particular file that is part of the build.)

The *basic* motivation is simply for a human to see a message ID in a log file; then find it directly in the code to find the originating algorithm/logic/whatever. File/line/function are much more likely to change over time.

<...>

Long story short: by allowing for negative and positive (allow-list/deny-list) filtering by message ID, messages seen in production can be limited to a low volume without sacrificing flexibility in diagnostics/debugging w/r/t daemons in the field. E.g.: INFO messages can be disabled but with an allow-list of particular message IDs; WARNINGs can be enabled but with a deny-list of particular messages which are valid error conditions but too verbose or useless in particular conditions. Crucially, the coder can continue with only a handful of well understood severities – instead of a zoo of 10 log-levels – while in the field they can be *dynamically* adjusted.

This presupposes the application supports dynamic config. flow.cfg supplies this for example.

Perf-savings-vs-log-usability story for <a given production daemon>

Briefly a slightly deeper dive into the thing summarized in the last paragraph:

Not that it is particularly surprising, but while profiling <a certain daemon>, people have found that logging figures strongly in the processor cycles used. It was not the bottleneck causing the main issue, but with that issue resolved: processor cycles become the gating factor for hardware use, and within <that daemon> the logging code is responsible for significant resource use in line with that. Namely:

- Very many lines are generated at the standard production log level (INFO; meaning INFO/WARN enabled; TRACE/DATA disabled). I've heard estimates of several gigabytes in 6 minutes.
  - This uses the storage hardware heavily.
  - It also means gzipping by while rotating is done frequently.
- A Flow.log log call site acts as follows.
  - The FLOW_LOG_*() macro checks the verbosity filter (basically, checking against a single verbosity level – INFO in production usually – but can be done per component enum value also). If it fails, done.
  - Otherwise the << part of the macro log call site is computed: A thread-local Boost.Iostreams string-ostream is looked up; then written to. This can be quite a real computation (the idea is to not log so frequently that it matters in the big picture). This occurs directly in the thread that wants to log.

        - This is *significant processor use synchronously in user thread*.
    - A task is post()ed onto flow::log::Async_file_logger's dedicated single async-logging thread, passing the assembled message plus metadata (line, file/function static string pointers, time stamp, severity, component enum, thread ID or nickname) as captures in the task. The async-logging thread, when it gets to this task ASAP (via normal Boost.asio task queue mechanism) executes it: Namely, it writes the stringification of the metadata and the assembled message to an `std::ofstream` (note: not `cout`, not `cerr`, but a file stream), capping with std::flush to actually write to the file (enabling `tail -F` use among other things).
        - This is non-zero processor use in the async-logging thread: and note that these tasks may be coming from tons of concurrently hard-working threads. (There is work planned to potentially speed this up by using FILE* instead of `ofstream` or even doing manual RAM buffering combined with direct native `write()`s.)
        - Naturally it's also heavy I/O (storage) writing.

The main insight – perhaps obvious – here is that there's real processor use and I/O, and *this significant use is directly proportional to how many filter-passing messages are, in fact, logged by the daemon.* So, if we logged very little, then the above resource use would be reduced strongly. Great! Let's not log!

Obviously this is in tension with many useful messages for diagnostics/debugging in everyday use – especially in the first months/year of the launch. Anecdotally speaking, messages being useful to understand X or Y in the field, is a common occurrence. Great! Let's log lots!

How to solve it for <a production daemon> then?

So the problem statement is, we want to log useful stuff... but the least amount of useful stuff possible, to save resources and enable efficient hardware use. So how to achieve this?

- First approach is to just increase the log level to WARNING (instead of INFO), so INFO messages don't show up. Even WARNINGs are not insignificant, so setting log level to NONE in config could even be advisable. Naturally, we will then lose all useful logging. Not great.
- Second approach is to tweak log level by component. (This feature is readily available in Flow.log. Enabling it in <a given daemon> is a <1 hour project; basically just configure a log::Verbosity_config instead of a log::Sev; it's even backwards-compatible in terms of the actual config file contents.) This does allow for much more flexibility; but some components alone log a metric ton of messages, so it's simply not fine-grained enough to solve the problem. Better but still not great.
- The other approaches involve Flow.log code changes (which is what this proposal is about after all).
    - The naive approach would be to just add another log::Sev; perhaps EVENT – between WARNING and INFO. Then, determine a bunch (not too many) of known useful INFO messages and change them to EVENT severity (FLOW_LOG_INFOFLOW_LOG_EVENT). And naturally set the log level to EVENT, not INFO.
        - This *might* solve the problem temporarily. There are however significant downsides:
            - This is not dynamically changeable. If we get it wrong, it takes a release – with code changes – to fix it. Moreover it cannot be easily applied to a specific machine, or region, without replacing a binary manually. Manual Changes are no joke in production.
            - This adds significant burden on the coder: Like today, when one hems and haws about making a particular new message TRACE or INFO: It's useful, and it's only one per stream, so let's make it INFO! Great, but now EVENT enters the picture, and the coder will have to decide whether it's INFO – not perf-*killing* but too verbose for production – or EVENT – important enough for production.
                - Eventually we'll just be back here again: Too many EVENT messages! Log size has creeped up! Longer-term code maintenance will become that much harder.
                - It's also a step on the road to the log level zoo with 10 possible values, where no one *exactly* knows what they mean. Adding a severity has a high tax-like cost over time. We don't want to go down this road unless absolutely necessary.
    - The non-naive approach is to use message IDs – hence this proposal.

Since that was written we did add some more log-levels by popular demand; Flow notes these are optional to use and should be avoided unless really desired.

So, now, suppose there is no new EVENT severity. However, suppose we set the log level to WARNING – meaning INFO messages are not logged ever (nor TRACE, nor DATA). Now, as proposed above, let's find a bunch of messages we find useful. Add their message IDs to the config file and dynamically apply it (to the network, or a region, or a machine). Boom, it takes effect immediately. Need more (need some specific info)? Dynamically update. Need fewer? Dynamically update. It's flexible/usable: and the coder is left unbothered throughout. And INFO can still be re-enabled wholesale if desired. For example in dev/dev-test INFO is a reasonable log level setting: it doesn't slow downloads to a crawl – just isn't as hardware-efficient as it needs to be in production. For another example, maybe we just want full INFO logging for a few minutes on one machine: no problem. Then get back to only specific messages.

Even certain TRACE messages can be enabled in a pinch. That's something I've definitely wanted at times, and it was frustrating that enabling TRACE wholesale would've made the service unusable in production – while doing so for an individual message would have been fine.

Furthermore suppose (as already occurs during checklisting, I have heard) some WARNING is undeniably a valid WARNING... but it's annoying or happens too often or ???, and we just don't want that one anymore. OK: In the same way that an allow-list of message IDs was used to enable certain INFOs, a deny-list of message IDs would be used to *dis*able certain WARNINGs.

(To be clear – I'll get into it in detail below but for now to put your mind at ease: these are only proposed conventions for <a particular daemon> regarding WARNING or INFO. Flow.log message ID system will allow flexibility to apply these allow-lists/deny-lists of message IDs to any severity or severities. I am just explaining the use case.)

So that's the intuition.

The design sketch – in Flow.log – sans filtering

Now to describe the actual feature; but in this section without the filtering-by-ID feature yet.

Take FLOW_LOG_INFO() (the same applies to all the others, e.g., FLOW_LOG_WARNING/TRACE/DATA...). That API shall remain unchanged. The coder shall be expected to just write a log call site as normal without thinking about message IDs.

Add (usually not used directly) API macro FLOW_LOG_ID_GENERATE() (no args) which evaluates to simply `0`.

Add new API FLOW_LOG_INFO_ID() which takes an extra argument at the front: the message ID, a uint16_t. Otherwise it is identical to FLOW_LOG_INFO(). Internally, FLOW_LOG_INFO(X) is equivalent to FLOW_LOG_INFO_ID(FLOW_LOG_ID_GENERATE(), X), and 0 is a reserved null ID (real IDs start at 1). Internally, easy peasy:

- The message ID (including 0) is added to the bits of data in Log_msg_metadata (time stamp, severity, file, line...) and is passed around along with those. It's pretty small at least and fine to copy.
- When Ostream_log_msg_writer actually writes to an `ostream` (`ofstream` in the case of Async_file_logger), it will output the message ID just preceding the message proper. If 0 maybe omit it. If not *maybe* omit file /function/line (or make that configurable in log::Config; TBD; details).

We will need a script/program, perhaps named **update_message_ids**, which shall modify the source code. It will replace each FLOW_LOG_INFO(X) with FLOW_LOG_INFO_ID(N, X) and related work. Details:

- update_message_ids shall act on each module (in the Flow.log sense; so Flow itself is a module; <some daemon> is a module; etc.) separately.
- It shall maintain a file (maybe **update_message_ids.state.json**) in the module's directory structure storing the max message ID used so far (or maybe that +1). If no file is present the max ID so far is 0.
    - After finishing, it shall create or update the file and save the new max ID.
- It shall do these things:
    - First, it shall convert all FLOW_LOG_INFO(X) (etc.) to FLOW_LOG_INFO_ID (FLOW_LOG_ID_GENERATE(), X).

    - Next, it shall convert all FLOW_LOG_ID_GENERATE()s – both ones used directly by user (in connection to should_log(), but that's discussed in later section) and ones just added in the above step – to newly generated IDs.
- It shall be quite careful about formatting.
    - It needs to omit comments and such things and find the *actual* log call sites, and it needs to accurately find the start *and* end of a given call site.
    - To ensure prettiness *and* not making the file wider (column-count-wise), perhaps format each replacement call like this (note that all original content, at worst, shifts to the left (with one ultra-minor caveat, omitted):

```
FLOW_LOG_INFO_ID // Preserve comments and such.
  (<generated ID>,
   <original line 1> // Preserve comments and such.
     <original line 2 if any -- indent 2 to the right per Flow style guide>
     <original line 3 if any -- ...>
     ...);
```

Caveats:

- FLOW_LOG_<...>_WITHOUT_CHECKING() (and similar) is a corner case. It always involves a nearby should_log().
    - The script shall not touch such a macro but shall fail to execute and print an error message with instructions about what to do. This is discussed in the below section on message ID filtering.
- Is it possible for the coder to *not* associate a message ID with a log call site? Yes.
    - If update_message_ids is never applied to a particular module, then everything remains as today. There's no problem.
    - Otherwise, coder *may* still (not sure why but...) simply pass in the literal `0` and manually invoke FLOW_LOG_..._ID(0, X). Then update_message_ids won't touch it, as it involves neither FLOW_LOG_...() nor FLOW_LOG_ID_GENERATE(). It has explicitly chosen to log with the null ID. So then presumably most log call sites will be converted automatically to use a message ID; and if some site does not want to, then it can explicitly use 0.

Implementation note: <a certain daemon> logging already has this feature and hence has their update_message_ids script counterpart. Dirk Jagdmann says in a chat message:

- In the <codeline> I've rewritten the programs, originally from Ankur, to assign those unique IDs. <...>

- So if you're looking into some program to change the .cpp source code and assign unique IDs, you should look into that.

The design sketch – in Flow.log – filtering

Today Logger::should_log() (pure) API takes only the Sev and the Component. In this design we also add the optional ability to use an allow/deny-list of message IDs. This also affects the following aspects of Flow.log:

- flow::log::Verbosity_config – which is how one can specify a verbosity policy with a string – will need a way to specify these allow/deny-lists.
    - Accordingly flow::log::Config will need to support applying such a policy either manually or via a Verbosity_config (like today, but with the allow/deny-list feature as well).
- update_message_ids (described above) will need to account for the corner case wherein a coder combines a manual should_log() call with a FLOW_LOG_INFO_WITHOUT_CHECKING() (etc.) macro (presumably for performance).

Let's discuss should_log first(). Update Logger::should_log() (pure) API to take 1 more arg: message ID integer. The contract changes to saying that should_log() can (if it wants) take this value into account when returning `true` or `false`. All out-of-the-box Loggers in Flow will forward it to flow::log::Config anyway, so the real logic discussion is about Config (and Verbosity_config subservient to it).

- Caveat: should_log() with only 2 args should perhaps be provided for modules that don't want to bother with any of this (will never run update_message_ids). One approach:
    - In the API, default the arg to message ID=0; and
    - in the contract stipulate that should_log() must be able to handle this value gracefully.
- Breaking code alert! The few places that do use should_log() directly *should* therefore be changed to match the new API.
    - For any module where update_message_ids will not be run (i.e., module that doesn't want to bother with any of this new feature):
        - Just leave them alone. The default noted just above will be used.
    - For any other module:
        - For every use of should_log() (followed by FLOW_LOG_..._WITHOUT_CHECKING(), invariably):
            - Pre-pend a line like `constexpr unsigned int MSG_ID = FLOW_LOG_ID_GENERATE();`.
                - Note: Hence update_message_ids will replace the latter with a generated ID.
            - In the should_log() invocation, add MSG_ID in the message ID arg slot.
            - In the FLOW_LOG_..._WITHOUT_CHECKING(X) invocation, change it to FLOW_LOG_..._WITHOUT_CHECKING_ID(MSG_ID, X).
                - Note: Hence update_message_ids won't touch it; but it'll use MSG_ID which shall be set to the proper value.

Finally let's talk about the verbosity config modifications needed for this feature.

The existing verbosity config design (in Config and accordingly Verbosity_config which sets that policy via a compact string) remains in force. Config::output_whether_should_log() should execute all that logic as normal, first. In addition 2 more sets of knobs then come into effect.

- Allow-list knobs: These are ignored if output_whether_should_log(), to this point, has decided to *allow* the message through (`true`) anyway. If it has decided to *deny* (`false`), however, then:
    - Message ID allow-list: List of IDs, across all modules (they may *not* clash; details below), such that if the message's actual ID is in that list, then the `false` (deny) becomes `true` (allow) after all.
- Deny-list knobs: Inverse of the allow-list. If the standard verbosity check has yielded *allow*, then there's a deny-list of IDs that can turn it into *deny* after all.

Subtlety: As noted earlier, each module has its own set of message IDs, but multiple modules can and do co-exist in a given application (at least, Flow itself can log – as can the application proper). Each module's IDs start at 1, then 2, 3, ... Obviously they can clash. Hence each message ID list is, conceptually, actually a map from a module specification of some kind to an unordered set of message IDs within it.

- Config::init_component_names() takes a (today) optional arg which is a string prefix to pre-pend to component string names to disambiguate clashing. (So if module A has UTIL and B has UTIL, then one can supply "a-" and "b-" respectively for that arg in that call when initializing those modules.) The overall API shall be somewhat rejiggered to, perhaps, make that arg not-optional; to not be allowed to clash with the value for the arg in other calls to the init_...() method; and to *additionally* now be usable in Verbosity_config strings to specify which sub-allow/deny-list of message IDs the configuring user means.
- Verbosity_config's string semantics can remain unchanged except to allow 2 more optional sections following the required component/severity section that exists today:
    - "|+"; then a list of semicolon-separated sub-allow-lists:
        - the module prefix as described in previous bullet point; then a colon; then
        - a comma-separated list of message IDs for that module;
    - "|-"; then similarly sub-deny-lists.
    - Example: "flw-async,WARNING;h3p-front_end_proxy,TRACE|+flw:443,112;h3p:113|-h3p:112"
        - flow module async component verbosity WARNING, h3_proxy module f_e_p component verbosity TRACE;
        - flow module allow-list = message IDs 443,112; empty deny-list;
        - h3_proxy module allow-list = 113; deny-list = 112.

The internal implementation (in log::Config) of this stuff follows how the lock-free per-component severity-verbosity mechanism works. However it is mostly simpler.

- A message ID is uint16, so the allow-list table for each module is 65,536 bits (8Ki bytes). Same for the deny-list. Store this as a bunch of relaxed-atomics, same as we store the per-component verbrosity-severities, except each atomic here will consist of conceptually interally-mutually-independent bits, and each module's table has a fixed number of elements, all meaningful (each bit corresponding to a message ID).
    - If we run out of message IDs for a module, we may need to use uint32 (note: use a type alias for message ID from the start!). We *cannot* then allow IDs to go up to 4Gi-1 (too much RAM needed to store the config); just double the max allowed message ID by adding one bit, doubling the size of the table. 8Ki 64k IDs per module; 16Ki 128k IDs; 32Ki 256k IDs; clearly the RAM use is fine, until a module becomes unreasonably gargantuan in terms of log call sites. (Note: Code this forward-thinkingly, so that the max constant can be increased (possibly also changing the message ID alias to uint32), followed by a recompile, and that's it.)
    - Maybe even start with uint32 and 9 or 10 bits per message ID, as a proof of concept and to avoid realistically having to ever bump it up?
- Config internal struct Component_config – today storing only that module's per-component severity config – can absolutely *delightfully* (go, me!) now also store the aforementioned (conceptual) bit-set, comprised of relaxed-ordering atomics (probably uint8s for simplicity? or uint64s for speed? or...?).
- Config::output_whether_should_log() – upon reaching the allow/deny-list-applying stage described above – will (as today) look inside the appropriate Component_config; then check the proper bit in the proper atomic for the actual message ID passed to the method. If it is 1, then reverse the allow/deny result from standard severity checking. If it is 0, go with the standard result (meaning the allow/deny-list had no effect). Fast!

In discussing with Eddy Chan I proposed the additional feature of allowing filtering by file+line (__FILE__, __LINE__). The motivation behind this was components/modules – like Google QUICHE – that can hook up to Flow.log but themselves lack message IDs in their own logging APIs (and we don't want to change their source code namely running update_message_ids on it). I started writing this up – it is eminently doable (add file and line args to should_log() and hence support in log::Config and log::Verbosity_config). Then I realized there is a performance cost that's potentially unacceptable; at least so far we've found it unacceptable in designing/implementing flow::log::Config's should-log implementation. A check of a set of component/file/line combos involves, at least naively, an unordered_set<> – and if it can be changed dynamically (which is half the point here), then a mutex is required. Message IDs can be stored in an array of atomics, but these combos (naively at least) cannot.

There is probably some way to do this but it "feels" like an advanced v2 type of thing.

So what about, e.g., QUICHE? The QUICHE logFlow.log bridge shall, for now, pass some *one* message ID for all QUICHE message IDs (or just pass 0). Its individual messages cannot be turned or or off, sadly, then. However, the entire component can be configured w/r/t its verbosity level as usual. So at least we can set all of that guy to INFO or WARNING, as needed, and accept the resulting log volume or diminished debug info respectively.

The same is true of any other such component for which we do not wish to generate message IDs. If in a v2 we can come up with an efficient/lock-free way of dynamically configuring the component/file/line set – then this problem will be resolved.

Lastly

See the comments below - Philip Lisiecki messaged me to say he likes the general approach and had some additional feedback. So don't miss that:

A couple features that might be useful:

1. Ability to gather and report stats by component,message-id, including suppressed messages. This may be too much volume for query, but it could go to a separate log or some such every N minutes. It could use lock-free thread-local counters to minimize the CPU overhead for each log call.
2. Ability to configure a rate-limited message id: Some messages might be useful to know they are happening and to have some examples, but some conditions might end up happening *a lot*, and we may not want to waste CPU, disk, etc., preserving every one. (The downside to rate limiting is that someone might assume that some specific "bad think happened in context Y" event *did not happen* because it did not appear in the

log, not realizing that log lines were dropped. Even logging "dropped N messages" is not sufficient to completely prevent that if someone is grepping for "bad thing happened in context Y": there will be logs for other contexts, which do not match the regex, and the "dropped N messages" line also does not match. I.e., someone can see the type of message being logged and assumed it is always logged... Perhaps an indicator that a line is subject to rate limit might be a clue that that log line will not always be logged?)