Log Everything All the Time

Thursday, August 30, 2007 at 9:01AM

Todd Hoff in Strategy

This JoelOnSoftware thread asks the age old question of what and how to log. The usual trace/error/warning/info advice is totally useless in a large scale distributed system. Instead, you need to **log everything all the time** so you can solve problems that have already happened across a potentially huge range of servers. Yes, it can be done.

To see why the typical logging approach is broken, imagine this scenario: Your site has been up and running great for weeks. No problems. A foreshadowing beeper goes off at 2AM. It seems some users can no longer add comments to threads. Then you hear the debugging deathknell: it's an intermittent problem and customers are pissed. Fix it. Now.

So how are you going to debug this? The monitoring system doesn't show any obvious problems or errors. You quickly post a comment and it works fine. This won't be easy. So you think. Commenting involves a bunch of servers and networks. There's the load balancer, spam filter, web server, database server, caching server, file server, and a few networks switches and routers along the way. Where did the fault happen? What went wrong?

All you have at this point are your logs. You can't turn on more logging because the Heisenberg already happened. You can't stop the system because your system must always be up. You can't deploy a new build with more logging because that build has not been tested and you have no idea when the problem will happen again anyway. Attaching a debugger to a process, while heroic sounding, doesn't help at all.

What you need to be able to do is trace though all relevant logs, pull together a time line of all relevant operations, and see what happened. And this is where trace/info etc is useless. You don't need function/ method traces. You need a log of all the interesting things that happened in the system. Knowing "func1" was called is of no help. You need to know all the parameters that were passed to the function. You need to know the return value from the function. Along with anything else interesting it did.

So there are really no logging levels. You need to log everything that will help you diagnose any future problem. What you really need is a time machine, but you don't have one. But if you log enough state you can mimic a time machine. This is what will allow you to follow a request from start to finish and see if what you expect to be happening is actually happening. Did an interface drop a packet? Did a reply timeout? Is a mutex on perma-lock? So many things can go wrong.

Over time systems usually evolve to the point of logging everything. They start with little or no logging. Then problem by problem they add more and more logging. But the problem is the logging isn't systematic or well thought out, which leads to poor coverage and poor performance.

Logs are where you find anomalies. An anomaly is something unexpected, like operations happening that you didn't expect, in a different order than expected, or taking longer than expected. Anomalies have always driven science forward. Finding and fixing them will help make your system better too. They expose flaws you might not otherwise see. They broaden you understanding of how your system really responds to the world.

So step back and take a look at what you need to debug problems in the field. Don't be afraid to add what you need to see how your system

actually works.

For example, **every request needs to have assigned to it a globally unique sequence number** that is passed with every operation related to the request so all work for a request can be tied together. This will allow you to trace the comment add from the client all the way through the system. Usually when looking at log data you have no idea what work maps to which request. Once you know that debugging becomes a lot easier.

Every **hop a request takes should log meta information** about how long the request took to process, how big the request was, what the status of the request was. This will help you pinpoint latency issues and any outliers that happen with big messages. If you do this correctly you can simulate the running of system completely from this log data.

I am not being completely honest when I say there are no debugging levels. There are two levels: system and developer. System is logging everything you need to log to debug the system. It is never turned off. There is no need. System logging is always on.

Developers can add more detailed log levels for their code that can be turned on and off on a module by module basis. For example, if you have a routing algorithm you may only want to see the detailed logging for that on occasion. The trick is there are no generic info type debug levels. You create a named module in your software with a debug level for tracing the routing algorithm. You can turn that on when you want and only that feature is impacted.

I usually have a configuration file with initial debug levels. But then I make each process have a command port hosting a simple embedded web server and telnet processor so you can change debug levels and other setting on the fly through the web or telnet interface. This is pretty handy

in the field and during development.

I can hear many of you saying this is too inefficient. We could never log all that data! That's crazy! No true. I've worked on very sensitive high performance real-time embedded systems where every nanosecond was dear and they still had very high levels of logging, even in driver land. It's in how you do it.

You would be right if you logged everything within the same thread directly to disk. Then you are toast. It won't ever work. So don't do that.

There are lots of tricks you can use to make logging fast enough that you can do it all the time:

- Make logging efficient from the start so you aren't afraid to use it.
- Create a dead simple to use log library that makes logging trivial for developers. Document it. Provide example code. Check for it during code reviews.
- Log to a separate task and let the task push out log data when it can.
- Use a preallocated buffer pool for log messages so memory allocation is just pop and push.
- Log integer values for very time sensitive code.
- For less time sensitive code sprintf'ing into a preallocated buffer is usually quite fast. When it's not you can use reference counted data structures and do the formatting in the logging thread.
- Triggering a log message should take exactly one table lookup. Then the performance hit is minimal.
- Don't do any formatting before it is determined the log is needed. This removes constant overhead for each log message.
- Allow fancy stream based formatting so developers feel free to dump all the data they wish in any format they wish.
- In an ISR context do not take locks or you'll introduce unbounded variable latency into the system.

- Directly format data into fixed size buffers in the log message. This way there is no unavoidable overhead.
- Make the log message directly queueable to the log task so queuing doesn't take more memory allocations. Memory allocation is a primary source of arbitrary latency and dead lock because of the locking. Avoid memory allocation in the log path.
- Make the logging thread a lower priority so it won't starve the main application thread.
- Store log messages in a circular queue to limit resource usage.
- Write log messages to disk in big sequential blocks for efficiency.
- Every object in your system should be dumpable to a log message. This makes logging trivial for developers.
- Tie your logging system into your monitoring system so all the logging data from every process on every host winds its way to your centralized monitoring system. At the same time you can send all your SLA related metrics and other stats. This can all be collected in the back ground so it doesn't impact performance.
- Add meta data throughout the request handling process that makes it easy to diagnose problems and alert on future potential problems.
- Map software components to subsystems that are individually controllable, cross application trace levels aren't useful.
- Add a command ports to processes that make it easy to set program behaviors at run-time and view important statistics and logging information.
- Log information like task switch counts and times, queue depths and high and low watermarks, free memory, drop counts, mutex wait times, CPU usage, disk and network IO, and anything else that may give a full picture of how your software is behaving in the real world.

In large scale distributed systems logging data is all you have to debug most problems.
So log everything all the time and you may still get that call at 2AM, but

at least you'll know you'll have a fighting chance to fix any problems that do come up.
