Building a 100K log/sec logging infrastructure

David Lang - Intuit

Abstract:

A look at the logging infrastructure that one division of Intuit built that included the requirement to handle 100K lines of logs per second with the logs being delivered to several destinations (including proprietary appliances).

This paper will cover the options considered, the choices made and the problems we ran into. The most unusual and interesting topic discussed is the method selected to distribute the logs to all the different destinations, able to deliver the log message to several different load balanced farms of servers with only one copy being sent over the wire.

Introduction

Digital Insight (acquired by Intuit in 2007) built a new logging infrastructure in 2006. It replaced the prior logging solutions that had been implemented by individual teams that had then evolved . We provide Internet banking services to approximately 2000 banks and credit unions, providing everything from the Internet facing home-page to the connection to the bank's main systems. Over time and acquisitions, the environment had became very complex with 135 different networks, approximately 300 firewalls organized into about 100 sets, and over 1000 routers and switches. We needed a logging infrastructure to support the Security/Audit requirements of being able to go back in time and figure out what happened even months after the event when someone complained and as we looked at requirements we found that something that would satisfy these requirements should be very close to satisfying operational requirements as well, so we enlarged our scope. The final requirement document can be summed up as:

- Gather all logs generated by any software or hardware in the company.
- Have an alerting engine that generates alerts based on individual or combinations of log messages.
- Allow for rapid ad-hoc searching of the logs, both for fraud investigations and for troubleshooting.
- Maintain an archive of logs for many years (data retention policy set by the Legal department and driven by the need to provide logs of financial transactions to banks)
- Generate periodic reports summarizing data in the logs
- Be able to run for at least three years without needing any architectural changes. Proactively identify what the expected bottlenecks would be, and produce plans to address them.

At the time we had been growing rapidly, with the historic Internet traffic and log traffic approximately doubling every year for the 7 of the prior 10 years. As a result we estimated that to have the new logging system be able to last at least three years, it needed to handle approximately 100K logs/second.

In the following years the peak traffic and logs 'only' grew at about 60% per year, and the traffic became more concentrated, so the total log volume has only grown about 40% per year. In addition, about 3/4 of the possible logs still aren't getting fed into this system due to other priorities preempting the logging work, so the log volume did not grow to the design level. There have been peaks of logs where we have received >92K logs in a single second and the architecture has held up as expected. Normal peaks routinely generate 30K logs per second.

Architecture Design and Software Selection

The thinking for this project started in 2005 when we considered just adding Sensage to our existing logs to make it faster to search them. Initially Sensage quoted us a price, but when we went back to them a while later, they revised their price up by a factor of 4x. As a result, this moved from being a simple department purchase of software to a $1M USD project requiring much higher levels of approval.

We created a RFP and sent it out to several vendors, but in addition I was assigned to evaluate the options to build a vendor-neutral solution ourselves. This vendor-neutral solution could use whatever software we wanted, but it needed to be designed in a way that we would not be dependent on any one vendor. Any vendor's product could be swapped out for a different vendor's solution for that component without having to make major changes outside of the parts being changed.

We evaluated Arcsight, Sensage, Splunk, Nitro Security, and Greenplum as possible vendors. After evaluating the different responses to the RFP we decided to go with the vendor-neutral option. At the time only Arcsight claimed to be able to meet our requirements, but they wanted to charge us for each website that we host, which resulted in a pre-negotiation price of $40M being quoted. The other vendors either couldn't handle the 100K log/sec load, or couldn't handle the alerting/reporting functions. Splunk came the closest among the other options, but at the time their alert capability consisted of scheduling saved searches.

The architecture we ended up settling on uses *rsyslog* for the log gathering and transport, delivering the logs to multiple farms of servers simultaneously, with each farm focused on providing a specific set of functions.

Gathering Logs

We started by saying that most of the logs that we needed were already generated through syslog and so we decided to see if syslog could satisfy our requirement for log transport. Since many of our devices generated UDP syslog, there was a strong desire to be able to be able to support UDP syslog at least on the initial leg of logging, switching to TCP syslog or other protocol only if UDP was not reliable enough.

To test the syslog implementations, I created sample log messages using logger and captured the resulting network traffic with *tcpdump*. Then using *tcpreplay* to replay the traffic we performed tests with *sysklogd*, *syslog-ng*, and *rsyslog*. The tests consisted of setting *tcpreplay* to re-send the known traffic at a set speed. We then examined the logs written by each syslog implementation to see how many log messages were received. We then ramped up the message rate. This was not a formal study, but the rate of message loss grew so rapidly that a graph of the percentage of logs that were received would have looked like a cliff, so we just started talking about the approximate message rate at which the collapse happened. Over time most of the records of the testing have been lost so all that we have to go by is the records left in e-mail discussing the options. The initial responses were disappointing as none of them seemed to be able to handle anything close to the 100K logs/sec load we were predicting.

The *sysklogd* daemon we tested had already been tweaked (disabling name lookups, eliminating system calls and a streamlining a few other areas), but since it is single-threaded and had to write one message before starting to process the next, it started losing logs at a couple of thousand logs/sec, gradually losing higher percentages of the logs as traffic volumes increased

Syslog-ng seemed to hit a wall around 1K logs/sec and just dropped messages above this rate. I never was able to understand this. When I asked for help, the general answer that I got was to just use TCP.

Rsyslog handled short peaks up to about 30K logs/sec as it processed the incoming messages into a memory queue, but could only write out a few thousand logs/sec, so if the traffic spike was longer than the memory could handle it started losing massive amounts of logs.

Since *rsyslog* was the best of the available options, I investigated what was going on with it in more detail and immediately noticed the huge number of system calls that it was making per message, including four *gettimeofday()* calls per log message to track the message as it moved through *rsyslog*. The maintainer, Rainer Gerhards, was very responsive to suggestions for changes and performance rapidly improved. As the performance increased, the bottleneck became the internal locking of the message queues. Intuit sponsored some development to to add the ability to process multiple messages at once to reduce locking on the internal queues and be able to do database inserts of multiple records at a time. and within a few months we were seeing tests where *rsyslog* was absorbing spikes of 378K logs/sec (effectively Gig-E wire speeds with the 256 byte log messages that measurements on existing logs showed to be about average for out environment) and writing to disk at >78K logs/sec with no

dropped log messages. Since that time performance improvements have continued with some people now reporting success with end-to-end tests over 1M logs/sec.

Transporting Logs

Due to the large numbers of networks that we have, and the fact that many of these networks were isolated by proxy firewalls, we decided to implement a set of syslog relay servers. We built syslog relay servers in HA pairs and put an interface for a relay server on 90 of the major networks and accepted the risk that unreliable UDP syslog messages may be blocked by the router choke points from the other networks.

It turned out that another use for these relay servers is to clean up 'non-standard' syslog messages. Many devices send syslog messages that don't quite match the RFCs and as such cause problems when they are parsed and analyzed in combination with other messages. One example of this is that Cisco routers can be configured to send logs in the format

<pri>timestamp IP tag: message

or

<pri>timestamp name : tag: message

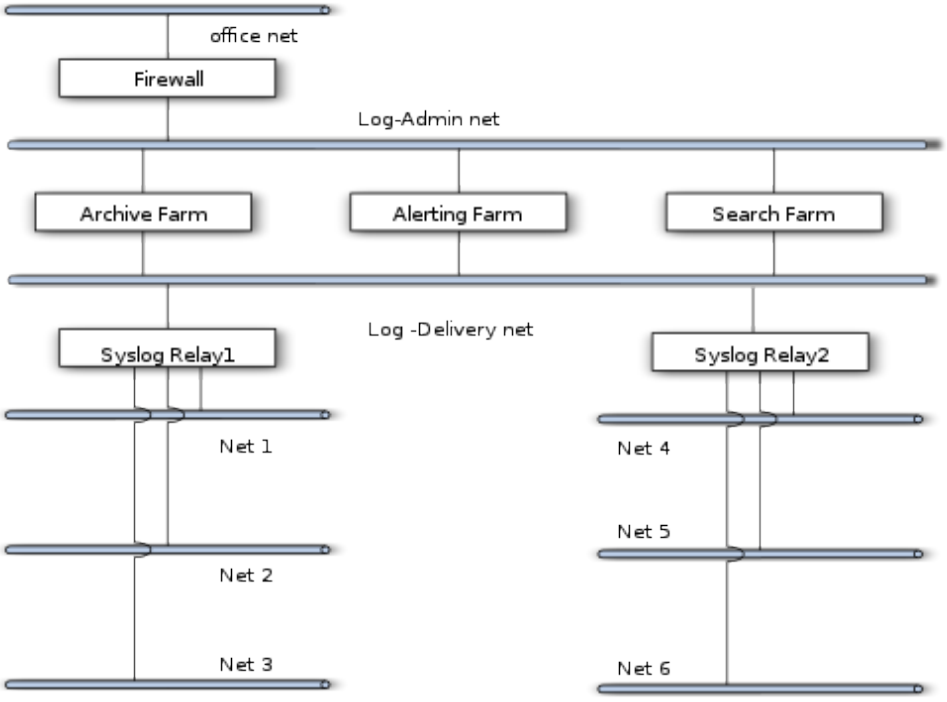

Illustration 1: Simplified Network Diagram

If you try to process these messages normally you end up with the syslog tag being just ':' instead of the %ASA-#-###### type of tag that is far more useful for filtering. What's worse is that if you have some Cisco devices configured to log by IP and some that log by name, you end up with the data fields misaligned on logs that are otherwise identical.

Initially we used the *rsyslog* custom formats and filtering logic to

detect and correct these log messages. Recent versions allow for custom parser modules to be written and loaded to fix the logs as they arrive on the system. I wrote and contributed several parser modules to deal with the broken logs we have encountered.

Delivering Logs

With the large volume of logs, we recognized early on that most of the processing of the logs would require more than one machine, and so we needed to figure out how to reliably deliver the logs to multiple destinations, and how to load balance across multiple servers for one destination. One issue that we realized early on was that at the data volumes that we were talking about supporting, sending multiple copies of the logs over the wire was going to start running into performance problems. With a goal of 100K logs/sec and Gig-E wire speed being just under 400K logs/sec, just sending four copies of everything would exceed network capacity. If we used a hardware load balancer, we would need multiple load balancers to avoid the interface on the one load balancer becoming the bottleneck. Moving to 10G Ethernet was considered as an option, but at the time the cost of of 10G Ethernet was several thousand dollars per port.

Instead we ended up using Multicast MAC traffic. Under Linux this is called CLUSTERIP. Normal IP Multicast uses the 224.0.0.0/8 address range and requires that the software involved be multicast aware. Multicast MAC on the other hand is a feature of Ethernet and works underneath IP. Multicast MAC sets the low bit on the high octet of the MAC address to 1 and the ethernet switches recognize this and send the traffic out to multiple ports on the switch. CLUSTERIP uses this by having the receiving system hash the connection information (one or more of Source IP, Source Port, Destination Port) and then divide the resulting hash into a number of buckets. It then passes any packets that match the buckets assigned to the local machine up the stack. This requires no changes in either the sending or receiving software. Everything is done in the network stack.

Example:

Let's say that we have the following three connections:

1. source 192.168.1.1 port 1025, destination 192.168.1.5 port 514 hashes to 13
2. source 192.168.1.1 port 1026, destination 192.168.1.5 port 514 hashes to 14
3. source 192.168.1.2 port 1025, destination 192.168.1.5 port 514 hashes to 15

We then say that there are 3 buckets in the current cluster, so we use the modulo function to assign these connections to nodes

1. hash 13 % 3 = node 1
2. hash 14 % 3 = node 2
3. hash 15 % 3 = node 3

The system that is configured to be node 1 of 3 then will process connection #1, the system that is configured to be node 2 of 3 will process connection #2, and the system that is configured to be node 3

of 3 will process connection #3. As long as you have a system configured for each bucket, all connections will be handled by some system.

This is done in IPTables with a command like:

```
/sbin/iptables -I INPUT -d 192.168.1.5 -i eth0 -j CLUSTERIP --new --clustermac
01:02:03:04:05:06 --total-nodes 3 --local-node 1 --hashmode sourceip-sourceport
```

This works with any sort of traffic (the port numbers obviously are only applicable to TCP or UDP traffic), with the only significant drawback being that all the traffic for the cluster will hit the kernel of each box before being dropped.

I recognized the fact that the switch distributing the traffic doesn't know or care which of the systems are actually processing the traffic. For connectionless protocols like UDP syslog, this then means that you can define your IPTables rules so that more than one box handles any given packet.

So if you configure two different systems with the rule:

```
/sbin/iptables -I INPUT -d 192.168.1.5 -i eth0 -j CLUSTERIP --new --clustermac
01:02:03:04:05:06 --total-nodes 1 --local-node 1 --hashmode sourceip
```

both systems will receive and process all traffic sent to 192.168.1.5.

In addition to this you can have the three machine cluster mentioned above, and each machine on that cluster will receive 1/3 of the traffic. There is no need to have the same number of boxes in each cluster, and the systems sending the traffic don't need to know how many boxes are in each cluster, or even how many clusters there are.

This allows us to add a farm of servers to the network by just configuring this multicast MAC on the system and then remove it later if desired without having any effect on the other systems.

By using this approach, the limiting factor is the port speed of the receiving boxes, in this case 1 Gb/sec or just under 400K packets/sec. This met our three year goals with about 4x headroom and the obvious upgrade path of moving to 10G ethernet when we ran into the limit (with the expectation that in 5 or so years when the limit would be hit, the cost of 10G equipment would have dropped significantly)

There were a lot of questions about the reliability of this approach, especially since it required us to use 'unreliable' UDP for the logs. As a result we ended up doing long-duration testing with multiple sources sending logs to multiple sets of destinations and we found that with a relatively low-end Cisco switch (3550) we were able to send several trillions of log messages in wire speed bursts of up 120GB per burst with zero lost messages as long as we paused between burst to let the slower file I/O that *rsyslog*

had at that time catch up. This testing was possible due to some machines with 128G of ram that were waiting to be installed, that we hijacked for a couple of weeks for this testing.

Since we wanted to avoid any possibility of other traffic interfering with the log delivery, we designed the core logging servers to be dual homed, with one network being used only for log delivery, and the other being used for queries and administrative traffic.

Log Analysis

With this capability in hand, we then built out several farms of servers to receive and process the logs.

Archiving:

We are using a simple *rsyslog* server pair writing to flat files that then get compressed and rolled to long-term storage (initially a 16x1TB RAID 6 array)

Reporting:

We wrote a series of scripts (currently a combination of Bash, Awk and Perl) to implement Artificial Ignorance filtering of the logs.

This consists of:

- Filter out messages, but count messages that are known to be uninteresting (the number of times an uninteresting thing happens may be interesting)
- Split logs which you recognize off to separate scripts to summarize them.
- Then sort the remaining messages by the most common messages and generate a report

This is still being done on the same farm as the Archiving. Periodic re-writes and optimizations have allowed the reporting to be completed in a reasonable time frame, but as the number of logs increases, this will eventually need more servers and we will then have to decide between creating a dedicated farm of servers for the reporting, or splitting the archiving across multiple servers.

Alerting:

We had significant disagreements internally over the choice of alerting tools to use (the “buy vs build” discussion). Management strongly favored of the “buy” approach but decided the marginal cost of buying two more servers to allow the people advocating the 'build' approach to implement Simple Event Correlator was small enough to allow us to both implement Simple Event Correlator and buy

Nitro Security's log analysis and alerting product.

We discovered that Nitro Security not only could not handle the load, but also that it could not handle the concept of syslog messages being relayed by other systems. It classified log entries based on the IP address the received packet came from, not the system name in the syslog header. Once it classified a log message, it only applied the 'applicable' analysis to the message. This meant that if the first message that arrived from a syslog relay box happened to be a Cisco ASA firewall, Nitro would decide that all messages from that IP were Cisco messages, and only apply the Cisco analysis rules to it. To try and work around this, I implemented and contributed a module that allowed *rsyslog* to forge the source IP address of UDP packets and implemented a set of systems to accept the messages and relay them to these appliances. While I was doing this, we implemented a work-around of implementing *rsyslog* filters on various other systems to relay logs of a specific type from a specific system. There were still serious performance problems, and even this work-around meant that Nitro believed that it had one system of each type reporting to it. Due to these other problems, we never got around to deploying the *rsyslog* forgery module in this environment.

We initially implemented Simple Event Correlator on a pair of 'high end' systems, 4x dual-core processors with 128G of RAM. Those boxes have kept up with the load with our current rule sets, so we have not needed to expand the farm. We have *rsyslog* filter logs of different types to different instances of SEC, each of which only sees a portion of the logs and only needs to deal with a portion of the rule set. This also has the advantage of allowing us to leverage additional CPU cores in spite of the fact that SEC is a single-threaded Perl program. We recognized that this approach had limited scalability, and we initially were thinking in terms of setting up memcached servers to allow for state to be held independently of the servers. However, in 2010 Paul Krizak from AMD submitted the LISA best paper “Log Analysis and Event Correlation Using Variable Temporal Event Correlator (VTEC)”[1] that talked about an approach of using syslog-ng to filter the logs going to several rule engines (perl scripts) and then taking the output of these rule engines and passing them via syslog to another rule engine for generating the alerts. Since then our scalability plan has evolved to follow a similar approach, using *rsyslog* to filter alerts by type, going to many instances of SEC, and then SEC setting global state by generating specially formatted syslog messages that could then be correlated by a 'master' SEC instance.

Searching:

For rapidly finding things within logs, we evaluated Sensage, Splunk, and considered rolling our own with Greenplum (a PostgreSQL derivative that can scale horizontally), but ended up deciding to use Splunk. Splunk provides a good user-friendly search interface with a pricing model of charging per gig of logs processed daily instead of per system that allows us to scale the hardware up to get better search performance without costing more than hardware. Sensage provided good scaling, but was very expensive and we would have had to write our own front-end for it. The PostgreSQL approach would have required us to become expert DBAs as well. Those of us with programming skills on the team had already shown that we were not particularly suited to user interface design.

1 http://static.usenix.org/events/lisa10/tech/full_papers/Krizak.pdf

Initially we built a by-the-book Splunk cluster with two forwarders receiving logs, feeding to a farm of four indexers, each with two dual-core CPUs, 16G of RAM, a mirrored pair of 300G 15K rpm drives, and a RAID 10 array of 10 1TB SATA drives. We ran in this configuration for a year and found the query performance was getting unacceptably slow, taking over 10 minutes for some queries to complete. We also had problems restoring data when one system got corrupted.

It was clear that we needed to upgrade our cluster, so we called in Splunk Professional Services and worked with them to do several weeks of performance testing on different hardware (making use of the year's worth of data and real-life queries that we now had).

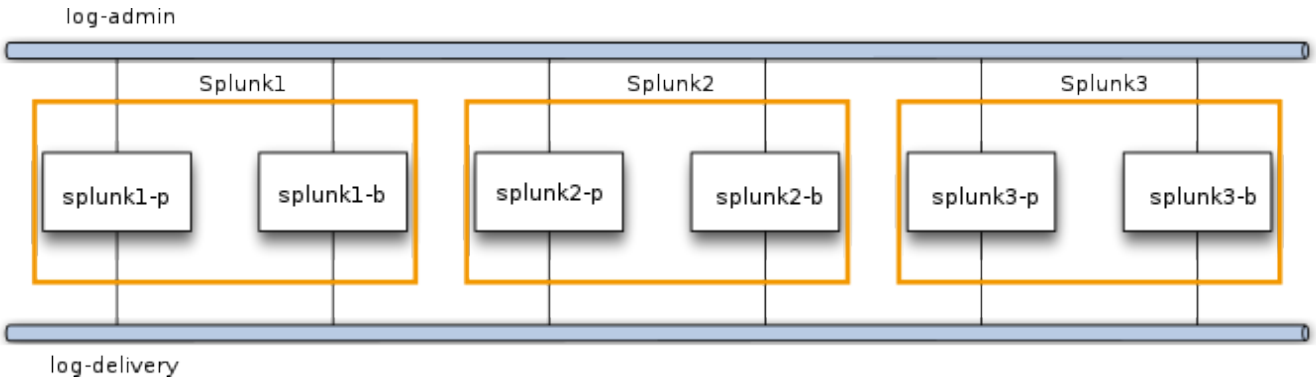

Illustration 2: Splunk Cluster Architecture

We also used their assistance to design and test (but have not yet fully implemented) a work-around for the lack of High Availability support in Splunk. The approach was to arrange the Splunk boxes into the equivalent of a RAID 10 array by configuring the machines into failover pairs. Each pair would be one logical node handling a slice of the traffic. Both the primary and the backup would be configured as the same CLUSTERIP node so they would both receive the same traffic. *Rsyslog* receives the logs and writes them to a file and every minute a cron job rotates these files and hands them off to Splunk. When it is time to roll the buckets in Splunk, each box checks to see if it is primary or backup of its pair. If it is primary, it copies its local copy of the bucket to the backup box, and if its backup, it deletes the bucket. This is not a perfect setup as the logs won't roll at exactly the same instant on both boxes of a pair, and it takes time to roll and copy the logs, so when a failure happens some logs will be lost or duplicated. The advantage of this approach compared to just letting the primary and the backup run independently (which would not have this loss/duplication potential) is that this keeps the log archives consistent between the primary and backup, so that after service is restored on a down box, its archives can be brought up to date with a simple rsync.

Another option that we considered, but haven't yet implemented, was to distribute the search load across both the primary and backup machines of a cluster. This would require running additional Splunk instances on both systems and having the bucket rotation script distribute the buckets across the

instances.

In our performance testing we discovered that for workloads that are close to read-only, as the Splunk searches are, RAID6 is just as fast as RAID10 as both can keep all spindles in use. We also upgraded the second-generation boxes to quad-core CPUs, 64G of RAM, a 64G Intel X25E SSD, and RAID 6 array of 16x 1TB SATA drives. The combination of the more powerful systems, as well as moving from four active indexers to 10 improved the performance of Splunk by about 16x. According to our testing, using the backup boxes for the search load will comfortably double the performance.

Lessons Learned and Problems Encountered

Overall this architecture has held up well. Other than upgrading the Splunk cluster, there have been no architectural changes to the base system. With the exception of normal patching, upgrading, and fixing failed boxes, the systems have pretty much "just worked". Other than the Splunk cluster, the only hardware upgrade that has been required was to upgrade the RAM on the archiving/reporting server as the working set for the nightly report had grown beyond the original 16G. Currently the box has 64G in it and the working set looks like it's about 40G.

The log reporting scripts have had extensive additions, and have been restructured or re-written when the log analysis runs start threatening to take more time to run than the time between runs. We've reworked them 3-4 times so far. and they have been a great example of "throw something quick-and-dirty together and worry about optimizing it later"

Since we have been using and implementing some of the latest features in *rsyslog*, this has resulted in us running development and git snapshot versions much of the time. This has occasionally resulted in a few problems and late nights, but overall things have worked well.

The only real technical "gotcha" we have run into is that the logs generated by the systems receiving the multicast MAC packets must not be sent to the common address. Otherwise, the Linux kernel "optimizes away" sending the packets over the wire, which prevents the other farms from receiving the messages. So the systems receiving the multicast logs must instead be set to relay-locally generated logs to one of the relay boxes for the relay boxes to bounce back to the farm address.

The multicast MAC logging approach has worked spectacularly well, and we've added and removed several products that people wanted to use over this time. New admins tend to be concerned about it, but then relax fairly quickly after they see it in operation for a bit.

The biggest problem has been the political balance between giving everyone access to logs that may

contain sensitive information, and letting people who can benefit from the system get access to it.

The security group had decided several years earlier to use Debian as our Linux distribution, and since we also used our standard image for firewalls, it was a very stripped down version which included a half dozen custom compiled packages for cases where we wanted different defaults, or where newer versions had features or bugfixes that we needed that weren't backported to the versions that Debian provided. The standard when we started was Debian 3.1 32 bit, and we moved to the 64 bit build very early on in the process and were running 64 bit kernels, even with the 32 bit builds.

Summary.

Creating a high volume logging infrastructure takes attention to detail, but the tools that are available and a standard part of all of the Linux Distributions now make it possible to build a logging infrastructure that can handle very high volumes of logs without locking yourself into proprietary solutions, and without writing a lot of custom, performance critical programs.

About the Author.

David Lang is a Staff IT Engineer at Intuit, where he has spent over a decade working in Security Department for the Banking division. He was introduced to Linux in 1993 and has been making his living with Linux (and using it as his desktop) since 1996. He is a Extra Class Amateur Radio Operator and served on the Civil Air Patrol California Wing Communications staff, where his duties included managing the state-wide digital wireless network. He has been running the wireless network at the Southern California Linux Expo since 2010. He is also active on various Open Source mailing lists. He can be contacted via e-mail at david@lang.hm, by phone at +1 818 292 7015

This paper and related materials are available at http://talks.lang.hm/events/LISA_2012/logging