rsyslog: going up from 40K messages per second to 250K

Rainer Gerhards
Adiscon GmbH
Mozartstrasse 21
97950 Grossrinderfeld
Germany
rgerhards@hq.adiscon.com

1 Abstract

Rsyslog is a modern syslog message processor. It has become the de-facto standard syslogd on most of today's leading Linux distributions. One of its design goals is to support a very high number of messages per second, especially on multiprocessor machines. The initial v4 versions of rsyslog did not reach this goal very well. In this paper, we describe the performance tuning effort we carried out to better match the goals. We describe the roots of the rsyslog project, typical (syslog) user perceptions and how they are related to program performance. Then, we classify the optimization done into four different types of efforts. Each type is described and weighted based on its overall contribution to the total improvement. Emphasis is put onto memory-subsystem and concurrency related optimizations, as these provided big benefit and seem often to be overlooked when optimizing user land applications.

While rsyslog is a very specialized application, it is our hope that insight into our optimizations, and failures of the initial design, is useful for optimizing other user land applications as well.

2 Introduction

Rsyslog is a system logger, a syslogd. From the average user's perspective, this means rsyslog will receive messages from local and potentially remote input sources, run these messages through a number of filter rules and write them to some final output destinations, most often files. So the traditional syslog perception (see figure 1) is rather simple. Note that most users assume messages are sequentially processed, that means one message is received, run through the filters and then written to the outputs *before* the next one is being processed.

Figure 1: Traditional user perception of syslog processing.

From the syslogd developer's point of view, a syslogd is simply a message routing engine. Messages received from the inputs are routed to their destination based on filter settings. This process can be compared to any message router and there is a strong analogy to network routers.

This analogy shows that syslog message processing is more complicated when looking in detail. Most importantly, there is not a single input, but potentially many, there may be very complex filter rules and there are also many outputs. Different inputs may want to send messages only to a limited set of filters, and thus outputs. Also, there is a high level of concurrency, both of messages from multiple inputs

as well as multiple messages from a single input as they are run through the engine. Also, outputs are not necessarily simple. They may be as complex as remote servers or remote databases. This means outputs may fail and may be suspended for an extended period of time. As in the case of a SMTP gateway, messages must then be temporarily buffered until the next hop can be reached. One thus can safely assume that figure 1 is a gross over-simplification.

However, this understanding is not usually present in user's perception of how the logging system works. Also, earlier syslogd implementations did not cover all these use cases. Most importantly, sysklogd did not support them.

We forked rsyslogd from sysklogd in 2004, initially only to add some limited functionality[1] that was considered too intrusive from the sysklogd project's point of view. After that was done, rsyslog was left dormant for about a year, and finally got momentum and an enhanced feature set. We must note that using sysklogd was important to gain something done quickly[2], but it also meant we needed to evolve based on its design, in careful steps. That obviously was far different compared to starting from scratch, where we would have been able to craft a totally new design. The original design we inherited was based on the simple user perception described above. Most importantly, it did not support any concurrency at all. So as rsyslog evolved, we needed to add various levels of concurrency.

Goals for rsyslog design and features were quickly added and then stayed stable. The rsyslog mission is to provide a logging solution that is easy to use for novice users while at the same time offering enterprise-class features. Among others, this means we aim at a very high message processing rate. To support that, it is our goal to scale well on multi core machines. The latter is considered vital as we anticipate massive multi core machines in the not so distant future.

2.1 Rsyslog Design

Rsyslog uses an object-oriented paradigm, but is written in plain C for many of the same reasons the kernel is[3]. Almost every functionality in rsyslog is provided by an object. Also, rsyslog is very modular: the rsyslog core is supplemented by loadable plug-ins, which implement many of the input, output, parsing and other important functionalities. Without plug-ins, rsyslog cannot do any useful work.

Plug-ins are put into some well-defined classes. For example, input plug-ins gather messages to be processed, output plug-ins consume messages and parser plug-ins parse message content. For each class, a specific *interface* exists, which must be implemented by the plug-in developer. Some entry points are optional. If not provided, a standard implementation is used instead (one may think of this as a class hierarchy and with inheritance). On the other end, almost all[4] core functionality is provided in form of class-like modules. A module (or plug-in) that requires access to the functionality first acquires access to the classes' public interface and then can manage class object instances[5] by calling the member functions[6].

A variety of plug-in interfaces and classes exist. For a rough overview, the input and output plug-in interfaces and the queue class shall be sufficient to convey the idea of how rsyslog is designed (figure 2): There exist potentially multiple input modules. Each of them runs on a separate thread[7]. Messages gathered are submitted to the queue engine, which is responsible for enqueueing *and* processing of

[1] namely native support for writing to MySQL

[2] It is questionable if rsyslog would ever have been able to succeed if we would not have the advantage of a "basically running" solution right from the start. So we still think that basing rsyslog on sysklogd was the right thing, even though it constrained rsyslog's design.

[3] Reasons like the ability to actually fine-tune code without any runtime overhead affecting the picture. For more details, see [1].

[4] some functionality is either not yet converted into class-type structure or will probably not be converted because of complexity associated or overhead involved.

[5] actually C structs

[6] We use function pointers to do this, much in the same way a C++ compiler does it "under the hood".

[7] As of the interface specification, input modules are permitted to spawn their own sub-threads. However, none of the rsyslog-provided inputs does this, and we also do not know of any third party input plug-in utilizing multiple threads for input gathering. So we can assume inputs always run on a single thread.

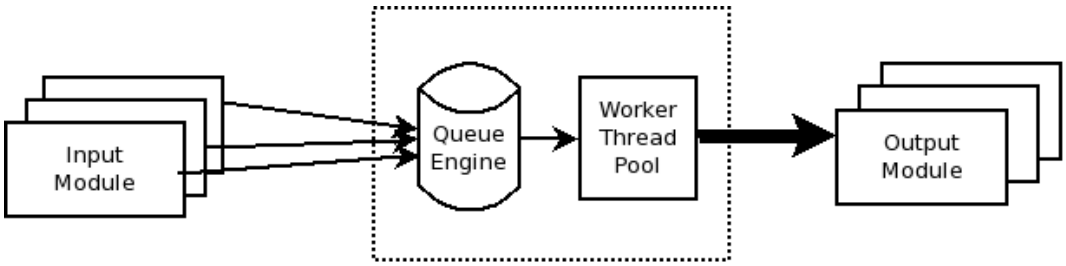

Figure 2: (Very) rough overview of classes in rsyslog. The box enclosing the queue engine and worker pool actually encapsulates a number of classes, including the filter functionality. The big arrow going out of this box to the outputs shall signify that a number of parallel activities hit output modules.

messages. To do so, it manages a worker thread pool, that spawns and controls a number of worker threads, processing different messages concurrently. The worker threads call the actual output. It is important to note that the output module itself must be reentrant, but a single output action is never called concurrently as of the interface specification. In that picture, concurrency is encapsulated inside the queue engine.

3 The Performance Optimization Project

Rsyslog is currently deployed in some of the world's largest data centers handling terabyte of traffic each day. A single early v4-instance could handle roughly 40,000 messages per second (mps)[8]. Obviously, that number was somewhat low for a high-demanding data center. Even worse, it turned out that rsyslog scaled very badly on multiple cores iff the filters and outputs were rather trivial. Adding cores could even lead to *decreased* performance. This was a design issue, and we will describe the reason in depth in section 7.

Based on practical needs, we initiated a performance tuning project in early 2009, which lead to coding mostly in the May to July 2009 time frame. The goal of this project was to two-fold:

- speedup processing of single messages
- permit rsyslog to process a larger input set as processors are added, ideally with a linear speedup

The first goal leads to improvements on both uni- and multi-processors. The second goal does not speedup processing of single messages but rather permits a larger overall throughput (at the price of utilizing additional resources). While the first goal was obviously important, the second goal was (and is) considered a key element to long-term success. Processor designers have reached an upper limit on the speed a single core can provide. Clock frequency is limited by physical properties and the amount of fine-grained concurrency a superscalar processor can extract from sequential programs is also limited. So processor designers now focus on providing parallel machines with multiple cores, exposing the concurrency available in hardware to the software layer. To work well with future processors, software needs to utilize this concurrency offering. In short: we cannot expect single cores to become much faster in the future, but we can expect many more cores inside a single CPU. So applications must be able to gain speed by parallelizing tasks.

The tuning effort was not meant to generate scientific results. Thus, unfortunately, we can not provide hard evidence on the precise effect each measure had. But in general, we were able to increase performance by more than 500 percent (on a multi-processor system) and know roughly which class of optimizations contributed which part of the improvement.

The optimizations we did fall roughly into four categories:

[8] mps based on typical, less than 100 byte messages and file output with a limited number of filters.

1. traditional optimizations, targeting to decrease the per-message processing time
2. code refactoring
3. memory-subsystem based optimizations
4. concurrency-related optimizations

Many of the individual measures taken fall into just one of these categories. However, some of them affect multiple categories. In this paper, we will focus on the above categories. To see the exact sequence of optimizations made, check rsyslog's public git server [6].

We initially did not specify the categories. Instead, we analyzed the work performed in order to gain "lessons learned" from it. During this task, we identified the categories. In the future, we will most probably use that knowledge to hopefully analyze performance bottlenecks, and potential solutions, quicker than before. Most importantly, we have seen that the impact on performance increases in the order given above. That is, concurrency-related optimizations offer far more potential for speedup than do traditional, control-flow oriented optimizations. We believe (but do not have hard evidence yet) that this holds true for most applications having similar properties like rsyslog: the processing of mostly independent, somewhat similar objects within a type of pipeline (a typical pattern found in servers of all kinds).

In the following sections, we will describe each optimization class and provide some details on solutions. Note that the list is not exhaustive. We focus on those things that we consider most valuable.

4 Traditional Optimizations

In this category, we reduced the time required to perform a specific function. In essence, this means improving algorithmic efficiency, together with environment-specific optimization. Tuning of this type is very common and many papers have been published about it, so we would like to just briefly describe some efforts.

4.1 C Strings vs. Counted Strings

One thing that really made a difference was the usage of counted strings vs. traditional NUL-terminated C strings. Initially, we needed to obtain size information by strlen() (due to the inherited and then evolved code). It was quite simple to replace that by counted strings, especially as in most cases we knew the exact byte count at string creation time. While this required an extra counter to be maintained (see section 6 for potential issues), the saving by far outweighted that.

4.2 Precomputed Constants

While this is very common knowledge and we always tried to precompute constants, code evolution lead to situations where constant-like objects were computed each time they were used. This is probably more related to refactoring (section 5), but we would like to mention it here as this seems to be more likely to be done in a traditional optimization step. It is relatively easy to identify such problem areas. One must check for C constants, being supplied to functions creating other objects (and as sole parameter). These are good candidates for precomputing.

4.3 Operating System Calls

One thing that we focused on was the number of operating system calls (OS calls). For many of them, a context switch is required, which is a very costly operation. A very good example of these were queries of the system clock (named time() for quick reference). Rsyslog needs time at various places, among others

1. reception time of a messages
2. current system time to build date-based file names
3. time-based filters (e.g. execute action only once every n seconds)
4. timestamp used to age some internal structures (caches, etc...)

In general time() is a very quick system call. However, if a large amount of messages is to be processed, and several time() calls are needed to process a single message, the execution overhead of these calls becomes an issue.

One of our users, David Lang, pointed out that he saw a too-large number of time() calls within straces he did. He suggested to reduce their number. We followed his advise and analyzed when and why time() calls were done. There were situations where in function a time was obtained and used and then function b was (always) called, which also obtained time. As this happened in sequence, there usually was no difference between the two timestamps. And even if there was a difference, it was not of real importance as function b just wanted to have a "current" timestamp. So we could simply avoid time calls here by passing down the already-obtained timestamp to functions called in sequence.

Other reductions were possible because we looked at the semantics of why time was obtained. When they were used to age structures, actual system time was not always necessary. Often, we actually just needed a monotonically increasing value[9]. So, in this case, we could simply replace time() by an integer value, which we incremented with each query[10].

As one final case, we identified situations where exact time was not strictly necessary. System clock resolution is not very accurate at the sub-ms level. In a tight receive loop (for example inside the UDP receiver), we often obtained the exact same timestamp for a relatively large set of messages[11]. For a TCP receiver, things were even more obvious: if a large reception buffer is used, we can receive several thousand messages right in the same instant. From the rsyslog application point of view, time of reception is when the OS placed messages into our app-provided buffer, and this happens all at the same time. This lead to the conclusion that we often were unable to get an exact notation of time. As such, we concluded that it often is sufficient to use a somewhat less accurate time, and spare time() calls by doing so. For example, in the TCP case we use a single time() query for all messages within the same reception buffer. For UDP, the user can configure how often time() is to be called. The idea here is that we do a re-query only every n messages, assuming that n is low enough so that time() will most probably return the same value even if we did a query for each message[12].

In some related cases, we could relax accuracy requirement to not operate on actual system time but on time fields already contained inside the message object (namely reception timestamp). For many cases, that was a close-enough approximation of actual system time.

Using a combination of these methods, we were able to dramatically reduce the number of time() API calls. That lead to a very notable reduction in context switches and thus a considerable speedup.

4.4 Buffer Sizes

Another traditional tuning method with good effect is increasing buffer sizes. For many inputs and outputs, increased buffer sizes mean decreased number of OS calls and thus improved performance. However, this is not possible in all cases. Looking at the input side, it works perfectly well for transports like TCP, where a single OS call can return multiple messages at once. Transport like UDP cannot benefit

[9] In one case, we even found out that our usage of a timestamp was inappropriate, because the resolution was not good enough to prevent duplicate values. Duplicate values could cause mild problems in that case.

[10] We used 64 bit values, where a wrap should not cause any problems at all. In one case, we used 32 bit integers on 32bit platforms because a wrap to 0 would not cause problems but rather a very mild performance loss —much less than doing atomic 64 bit arithmetic on those platforms.

[11] depending on message sizes, we could see this for a couple of hundred messages in some cases

[12] For obvious reasons, this logic is only used as long as messages are present in OS buffers. Once we go to a real blocking read, a time() call is always required when the next message arrives. This can be potentially much later.

from that, because there is no OS call that permits to read more than one message at one time. On the output side, it is a matter of user needs. For the file writer, we implemented a buffered writer approach, where rsyslog internally fills a write buffer and flushes it only after a non-activity timeout expires or the buffer is full. However, this means that files are no longer written as messages come in, but rather in buffer blocks. This was contrary to common user expectation, so by default we turned off that mode. In the future, we plan to enhance the algorithm so that we can use a very short inactivity timeout and the ability to define an additional timeout where a partial buffer is written if it is above a configurable mark. However, one must be very careful with such background actions: implementing them may force additional context switches, and this may turn out counter-productive and performance decreasing. So for our initial tuning effort, we decided not to pursue that route.

4.5 More Specific Algorithms

Rsyslog has grown rapidly and we tried to solve many problems by applying a single but configurable algorithm. This allowed us to move forward quickly, and also provides good readability and maintainability of code. However, for some use cases, the general code was non-optimal. For example, some rate-limiting functionality requires strict message sequence, thus putting a hard limit on concurrency, thus on scalability. However, it turned out that this functionality was almost never used. So we constrained the general algorithm just to handle a seldom-used border case[13]. We solved this by implementing multiple algorithms: the generic and slow one for complex cases, and a number of faster algorithms that could only be used if some constraints were not given for the configuration in question.

Using this method, we were not able to improve rsyslog's worst case performance. But we were able to improve the majority of real-world cases, sometimes very notably.

5 Code Refactoring

The rsyslog code has continuously evolved. New features, objects, and layers were added, functionality was moved between modules and objects and urgently requested features had to be introduced with only little overall design. Even worse, we started out with a design (sysklogd) that was not really suitable for what we had on our goal sheet. So the design itself needed to evolve as the code evolved. Also, open source projects like rsyslog often have no well-planned feature schedule and tend to focus on what is currently considered important or being contributed. This is not bad, as it ensures that the work reflects what is actually needed. However, it also tends to introduce more complexity than needed.

This was, and is, definitely the case for rsyslog. We have far too few code reviews and unfortunately there are very little external reviewers. As part of the tuning project, we did reviews of some code areas considered very important. Not to our surprise, we discovered that the evolved structure of interfaces and layers were much more complex than actually needed. For example, a higher layer reformatted some data element (a syslog PRI for example), and passed it down where one of the lower layers just undid that transformation and none of the interim layers needed it. The reason simply was that at a specific point in time a lower layer function l needed a specific format, which was than generated by the upper layer function u. As code evolved, l changed the encoding for some reason. It used the same format that u natively had at hand. However, at the time the change was made nobody thought about u. So l was modified to undo the transformation that u did, instead of removing the unnecessary transformation from l.

As another example, we saw that we had an extremely deep nesting of function calls in one instance. Analysis showed that many of these functions just passed parameters with little or no modification down to another function. These were not even interface layers, but rather object-internal functions whose processing had changed. We could combine some of these functions without loss of functionality and gained better readable and maintainable code.

[13] This specific type of problem was actually discovered during the second tuning project in spring 2010

These are the results one typically expects from refactoring. However, many people think that cleaning up code and restructuring it will possibly hurt performance. Maybe Fowler's famous quote [4] did it's part to this perception:

> "Refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning."

However, Mr. Fowler makes his position clear just one page later in this not-so-often cited remark:

> "I've found that refactoring helps me write fast software. It slows the software in the short term while I'm refactoring, but it makes the software easier to tune during optimization."

We fully agree with the latter quote. Even more, some refactoring immediately benefits performance. A good example is the reduction of call levels described above. Obviously, this directly improves performance. The same holds true for removal of unnecessary transformations. Demeyer has shown that even at the language-clarity level refactoring positively affects performance [2].

It is often important to not just look at code, but rather at the overall design as well. For example, we initially considered it useful for rsyslog to be able to shut down all worker threads when no work was to be done (after a reasonable timeout period). Analysis then showed that this resulted in a lot of complexity in worker thread pool management. Nearly half of the code was in some way involved with handling this requirement. Some was executed for each message being processed. Our initial goal for this mode was to save resources. However, it turned out that a blocking thread does almost use no resources at all. But shutting down a thread and restarting it requires comparatively large resources. So what was intended to be a resource-saver actually wasted them. After we understood this, we removed the capability and all code associated with it. Now, at least one worker is running, maybe in a blocking wait. The new code is faster, easier to maintain and most probably has less bugs in extreme border cases. This is an excellent example of the power that can be gained from refactoring and the proper code review that comes with it.

We consider larger-scale redesign efforts to be also an act of refactoring. For example, the current design of the network input/output layer and its sub-layers is considered suboptimal. It has grown too complex, is hard to maintain and has a worse-than-required performance. So while it looks decent from a purely academic point of view, it is over engineered from a practical perspective. Within one of the next development iterations, we will redesign this layer and hope to gain considerable benefit from that.

It should also be noted that development time is required in order to optimize things. Well refactored code saves development time. So with refactoring, the development team has more time to look at optimization. This obvious benefit is often overlooked, at least in our experience.

6 Memory-Subsystem based Optimizations

The memory subsystem has an enormous impact on execution speed. Unfortunately, algorithm analysis often does not include careful analysis of the memory subsystem. In many papers, memory is abstracted and access time to all memory addresses is equally fast and the same for reads and writes. Kamp recently pointed this out very clearly in [7]. He also told us that this does not mean algorithm efficiency reasoning is incorrect – but it does not apply as well to real computers as we thought.

Ulrich Drepper wrote an excellent paper on the memory subsystem [3], which can be consulted for all details. To understand the optimizations done in rsyslog, we provide a quick and coarse overview of the subsystem.

From the time when virtual memory (figure 3) was invented, we had a hierarchy: there was the physical main memory, initially with very fast and equal access time to all memory cells. But there was also the swap file, where memory chunks that did not fit into main memory were stored. Access to any cell in these on-disk memory blocks was magnitudes slower. It obviously makes a big difference if an algorithm could operate in main memory only or the disk. The less memory a process consumed, the

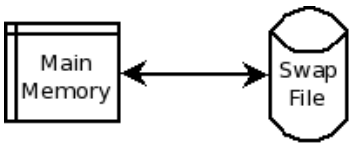

Figure 3: The traditional view of virtual memory: fast main memory is extended by larger but slower disk space.

higher the probability that it could be kept in main memory. An important concept is that of a processes *working set*. It is the minimum amount of memory needed by a process to carry out a closely related set of activities. In case of rsyslog, the memory needed to receive, filter and output a message can be considered the working set for processing a message.

As Kamp and Drepper note, the size of the working set has big implications on algorithm performance. For example, on a system supporting paging, there is a big practical difference between an algorithm that puts all required data structures onto one page in contrast to one that spreads data across many pages. The chance for page faults is much higher in the second case whereas it would be unexpected (but not impossible) to have more than one page fault in the first case.

Note that even with the simple paging VM model we do have different access times for read and write operations, though not immediately visible. With paging, the difference becomes visible only when a page is ejected: it needs to be written to secondary storage if and only if it was modified since it was last paged in. So a write, a modification, is more costly than a read, because it will force the page in question to be written on next ejection. We admit that this increased access time is hard to see and often irrelevant when looking at single writes because pages are ejected relatively seldom.

Also note that the worst case, asymptotical algorithm efficiency is not affected by memory operations. However, the real-world average execution time is affected. This mismatch makes it hard to prove the actual speed difference other than by measuring results.

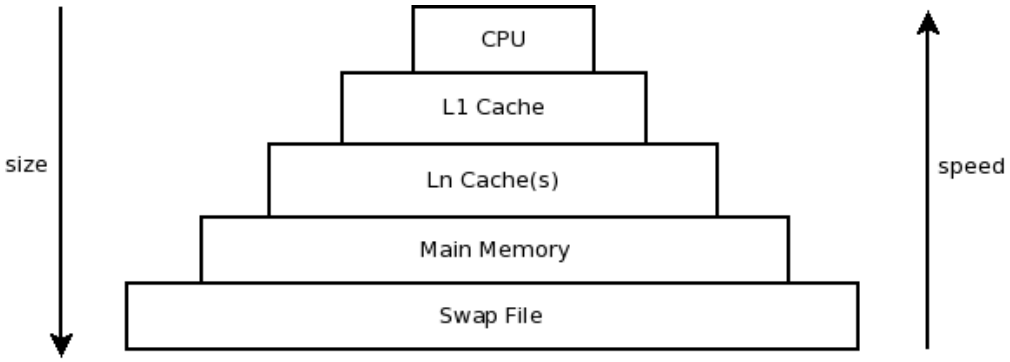

Figure 4: A modern memory subsystem: there are several level of caches between CPU and swap file. Lower layers have higher capacity, but slower access time.

We now need to make a leap from the old-days simple virtual memory model of figure 3 and move on to a modern memory subsystem, as shown in figure 4: today, there is a big difference between CPU speed and main memory speed. In order to keep the CPU working, multiple level of caches have been put between the CPU and physical main memory. Fast caches are expensive, and so caches of decreasing speed and increasing capacity are being used. It is hoped that frequently used data can be kept in high speed caches and so be accessed quickly. Main memory can now be thought of as just another, very slow and very high capacity cache. Transfer between main memory and the swap file still happens with large-size pages. However, transfer between main memory and the various cache level happens in much smaller increments called “cache lines”, typically 64 or 128 bytes in size. Cache lines are transferred as

a whole.

When a data element is read by the CPU, its complete cache line is transferred to L1 cache (if not already there). So any other data element inside the same cache line then can be fetched with almost no wait time required. This spatial locality has big effect on algorithm performance. However, this does not hold true if the cache line has been ejected from L1 cache. So temporal locality is also rather important, that is fast algorithms access nearby data elements in close temporal proximity. Algorithms doing that can perform much faster than algorithms that do not[14].

Even more important, modern CPUs typically have multiple cores. If so, caches must be kept coherent. This adds some extra overhead to writes. Also, as in the VM paging case, written-to cache lines must first be persisted to lower cache levels or main memory when being evicted from their current cache. All this adds considerable extra cost to memory writes compared to memory reads.

To sum it up, a modern memory subsystem has some properties programmers often do not think about:

- writes are (much) slower than reads
- access to data items in close spatial proximity is potentially much faster than access to items farther away
- access to data items in close temporal proximity is also much faster, what also means that recently accessed data is more likely to be accessed quickly again than data not being accessed for a longer period.

This has some direct practical applications to optimizing programs. The overall goal is to keep the working set as small as possible, and try to process everything that is related to a working set a before processing data in a different working set b. Also, it most probably is faster to avoid writes even if that requires the addition of some extra logic.

Finally, we need to think about concurrent activities: From early VM machines we know that the working set of all currently unblocked activities must fit into main memory to avoid thrashing[15]. While main memory thrashing is usually no longer found on today's machines, the same problem occurs at the cache level. Here it is harder to diagnose, because all of that is handled in hardware and we do not have any OS counters (like page fault rate) to detect the situation. Still, for good performance the working sets of all current activities must fit into L1 cache. So the number of concurrent activities place limit on acceptable working set size (and vice versa).

Inside rsyslog, we have short-lived and long-lived objects (and many in between, not to be elaborated about in this paper). The most prominent short-lived object is the message object itself. For short-lived objects, the optimal access strategy from a memory point of view is to create them, process them as quickly as possible without doing other activities and then destruct them. Ideally they should only be accessed for a very short period of time. However, it is advantageous to re-use the memory blocks occupied by these objects as often as possible. The reason simply is that this enhances the chance that they are still kept in fast caches, and repeatedly accessing the same memory keeps them there.

The primary goal for long-lived objects is a bit different: there are potentially many of them, and we often do not use some for prolonged periods of time. This means we can accept them moving to slower memory areas to make room for more frequently used items.

In all cases, spatial locality is quite important; for all objects, we would like to have all of their relevant data items into as few cache lines as possible, ideally in a single one. This has some implications for data structures. Traditionally, it is assumed that word-aligned data elements offer faster performance. At the extreme end, bit fields are said to be performance costly, because they require Boolean arithmetic in addition to address computation. However, packed structures and bit fields can be performance savers

[14] Drepper lists some impressive numbers e.g. for matrix calculation.

[15] Thrashing in this context means mutual ejection of working set pages by multiple activities because the main memory is too small to contain all required working sets at once.

if with their help we can manage to get a complete data structure into a single cache line. The advantages of high-speed cache access can by far outweigh the extra CPU workload[16]. If a structure is too big to fit into a single cache line, we must think about which data elements are usually accessed together and put these close to each other. Similarly, in a parallel program, it makes sense to think about which items are usually written to concurrently, and move these *far away* from each other, because otherwise they will cause cache thrashing by concurrent writes. There is one important implication from these considerations: often, data structures contain dynamically sized elements (namely strings). A typical implementation allocates these elements separately and places pointers to them into the data structure. This method is the best way to assure data is placed into *different* cache lines: the C malloc subsystem has some extra management information in front of the allocated data. So this alone prevents the malloc'ed regions from being packed. Especially in a multi-threading environment as rsyslog, malloc may allocated close-by memory chunks to different threads, thus spreading data elements over even more memory addresses. Another downside of this method is that pointers to the malloc'ed space must be stored inside the structure. On 64 bit machines, this means 8 bytes need to be written for each pointer.

Malloc also has some other problematic aspects: in order to work, it must maintain a free list and it requires time to allocate from that list and deallocate malloc'ed data again. This is especially inefficient for data that is frequently malloc'ed and free'ed, a scenario typically found in short-lived objects.

With that background information about the modern memory subsystem, we can now explore which optimizations we did in rsyslog.

6.1 malloc vs. calloc

This is the most basic of all optimizations. In some circles, it is considered good style to calloc all data structures. The idea here is that this will guarantee consistent initialization. There are a number of subtle issues with this approach, which we will not explain in detail here[17]. From a performance point of view, calloc means costly writes to everything inside the structure. This does not make much sense if most of the elements will be re-initialized to non-zero values right after the calloc. For rsyslog, we have closely evaluated all callocs() and replaced those where it made sense with manual initialization of only those data items that actually need initialization.

6.2 Using Stack instead of Heap Space

We reconsidered using stack memory over heap memory. The primary motivation was that malloc/free overhead is removed with stack memory. However, there is one important drawback from a correctness and debugging point of view: pointer bugs are always hard to find in C programs. For heap memory very good tools (like valgrind) are available. The toolset for finding stack based bugs is far more immature, if not non-existing. Also, pointer bugs hitting the stack can lead to very strange behavior, and are, in our experience, even harder to find than heap pointer bugs. We still favored stack over heap in almost all cases, because the performance benefit is comparatively large.

One big problem with stack-based memory is that it is not easy to do dynamic allocation in a platform-neutral way[18]. As such, we needed to bound the size of our data structures. Even though we reserve ample of (virtual) stack (address) space, we cannot cover any unusual use case. As a compromise, we allocate buffers of sizes that we consider to cover the majority of use cases. Then, we check if the allocated size is sufficient for the actual request. If it is, we go ahead and use stack memory. If we need more memory, we fall back to malloc and allocate it dynamically. That way, we can support very extreme configurations, while at the same time utilizing fast stack memory for the majority of cases[19].

[16] This becomes especially true when thinking about how many idle execution units a current superscalar CPU has under normal operations!

[17] One of them is that developer's often think calloc will necessarily initialize all pointers to NULL. While this nowadays nearly universally is the case, it is not true for all platforms. Another issue is that zeroed memory is probably not the best choice to detect program bugs.

[18] Glibc explicitly recommends against using alloca() and lists a number of known problems with it.

[19] To cover extreme environments, we also have made the fixed buffer sizes compile-time configurable. As such, advanced

6.3 Structure Packing

We have reordered and packed data items inside structs and kept access patterns on our mind while we did so. We also reduced the size requirement for many data items. For example, Boolean values were used to be represented by integers. We have reduced this to single bytes. We have usually not reduced it to bit fields because some of these items need to be passed by reference. While it would have been possible to convert this to bit fields, the resulting code would have been hard to read and complex. Here, we preferred better code readability over the ultimate in performance. We did this in the hopes to receive long-term benefits due to the simpler code (reduce the need for refactoring).

6.4 Reducing Malloc for Structure Data

We have a couple of data structures, namely the important message object itself, where we need to keep strings of largely different size. In earlier releases, this was done via the usual pointer approach, where we dynamically allocate memory for the element and store a pointer inside the structure (problems outlined above).

We tried to avoid this by supplying buffers supporting average sizes directly inside the objects. This comes at a price: the data structure itself will grow considerably, even in cases where the actual value requires a far smaller buffer. Also, when the actual value is too large, we fall back to dynamic allocation. That means that the in-structure buffer is not used at all. However, the potential reduction of malloc calls outweighs these negative effects, at least if the static size is carefully selected.

In one class of cases, there is no drawback at all (or a far limited one). A very good example is the syslog tag, a usually very short string somewhat identifying the syslog message. Tags are typically less than 16 bytes, often less than ten[20]. The pointer size on a 64-bit machine is 8 bytes. For tag and similar items, we create a union, where the pointer and the actual value use the exact same memory location. As we have byte-counted strings, we need to store a byte counter in any case. We use the byte counter to check if the actual data fits into the provided buffer. If so, we access the buffer area as data. If the string is too large, we access the buffer as a pointer that points to the actual data. For strings of up to 8 characters (7 if a NUL-byte is still required) there is no memory overhead. Even for tag, we have a very limited memory overhead. The only cost that must be paid is lightly more complex logic to make the addressing decision.

6.5 Reuse Memory Regions

We have reused some dynamic memory areas. The most prominent example is the template generator. It builds strings passed to the outputs for further processing. One template must be generated for each output action, so the code is frequently accessed. Also, templates usually require many memory writes, and thus are performance intense.

Before the tuning effort, the template generator did allocate an initial buffer on the heap, expand that buffer as need arises during template building (expectedly very seldom), passed it to the output in question and then discarded and freed the buffer again. Due to multi-threading, it was likely that some other thread obtained the same memory block before we allocated it the next time.

The tuning effort changed that. Now, the buffer is only allocated once per action, for the whole lifetime of the daemon instance. The very same buffer is used to build new strings. It is expanded, if needed, but never shrunk again (we consider this to be unproblematic for the syslog use case). The template is then passed to the output (as before) but not discarded after that.

This change provided two benefits: the number of malloc's was dramatically reduced because after an initial warm-up phase the buffer almost never needed to be expanded. Secondly, we reuse the same memory area ever and ever again. This makes it more likely that it will stay in high speed caches.

users can change these buffer settings should they not match their needs

[20] In practice, RFC3164 [9] and RFC5424 [5] limit tags to 32 characters, but some real world applications ignore that. In rsyslog, we have a default hard compile time limit of 512 bytes. Still, tag length over 16 bytes is typically very seldom used.

Similar modifications were applied to some other code areas as well.

6.6 Introduction of so-called Properties

There were a couple of data elements that stayed the same for a large number of messages, but needed to be different from occasionally. A good example is the remote host name for TCP connection when receiving messages. This value is obviously not a constant, but will not change during the lifetime of a TCP session. In typical syslog deployments, the value can remain the same for several hours and millions (or far more) of messages.

Before the tuning effort, the name was stored inside a connection-specific variable and then copied over to each message. This was not bad when the name is short (pointers usually require an eight-byte write as well). However, for longer names this is inefficient. And it is especially inefficient if dynamic memory must be allocated for it. This was the case in early v4. The reason is that when a TCP connection was closed, the memory buffer with the connection name would need to be destroyed, but at this time messages still referencing it could exist inside the system.

We looked at ways to solve this. One approach would have been to create a cache of names, which would never be discarded. That sounded dangerous and inefficient. Also, it was no general solution for similar problems.

We finally settled with a separate object type ("property"), basically a reference-counted string. At TCP connection creation we obtain the system name and create a property. Its reference counter is set to one and the name is copied over to it. Then, a pointer to the object is stored in the TCP connection entry. For each message received, we copy this pointer to the message object and increment the property's reference counter. When the message is destructed, the properties destructor is called. It decrements the reference counter and actually destructs only if it reached zero. That only happens when the TCP connection has been closed and all messages using the property have been destructed.

The approach helps reduce the working set because we will always refer to the same property for a large number of messages. Especially in message bursts this most probably means the property is always present in high speed cache. Reference counting is comparatively cheap. We use a 4 byte counter, so we need to update 12 bytes in each message to set the remote name. This is a bit more than the average local host name, but usually far less than a fully qualified domain name. Finally, we save the malloc/free effort otherwise required, a very considerable saving[21].

This method works very well in practice. We have applied it to a number of other objects and data items as well. However, it must be said that this method only works if properties are either not used by concurrent threads, or there is an efficient method available to do the reference counting in an atomic way. We have used atomic increment and decrement operations, which solve this issue with very acceptable performance.

7 Concurrency-related Optimizations

Many syslogd designs of the past were either single- or dual-threaded. Even single-threaded designs worked quite well, because all outputs were rather fast[22]. Rsyslog initially also had a single-threaded design. However, addition of (slow) database outputs clearly showed that at least decoupling of input and output was required. So v1 started by providing this dual-threading approach, an optional compile-time feature at that time. A big problem with that design was that messages were run in sequence through all outputs. If one output was stalled (e.g. a TCP or database connection being retried), all messages needed to wait until that situation was solved. The next problem was that we wanted to go modular, enabling anyone to write plug-ins providing functionality. It would have been hard to integrate input plug-ins into a single receiver thread. Also, we knew that with two threads, we could utilize two CPU cores at most.

[21] But we need to mention that we would probably have been able to reduce that as well by using expected-size pre-allocated buffers, as already described.

[22] Nevertheless, there often was some loss of UDP messages, most often not noticed by users but still existing.

This concerned us much, as at this time we already understood that future hardware would be massively parallel but a single core not much faster than today. So in order to scale well, we needed a totally different threading model.

To address this, we designed a new input interface, where each input runs on its own thread. These threads were somewhat heavy, as they contained message parsing functionality. Also, we designed the queue engine and it's worker thread pool manager. This was a key component to both provide potentially massive multithreading as well as decoupling actions. Starting with v3, there exist multiple queues, potentially one for each action and one so-called main queue, where all inputs sent messages to. The action queues could be used for slow actions, which than ran asynchronously to the rest of processing. This solved the problem of blocking or otherwise too-slow actions. The main queue was the primary provider of concurrency: the idea here was that multiple queue workers (upper limit user configurable) consumed messages from the queue, applied filters on them and passed them to the relevant actions. In theory, concurrent workers should be able to utilize highly parallel hardware.

Please note that at this time we still had the typical user perception of syslog sequence. We assumed that it was important to preserve message sequence, and this was the primary reason for a single main queue. We assumed that there was a strict order in which messages entered the system and we wanted to preserve that order all the way through the output, so message sequence as written in output files should be the same as reception sequence. Of course, multiple workers could cause some message reordering. This is an inherent problem which cannot be avoided if concurrency is desired. If even slight reordering was not acceptable, the user could turn off output concurrency by setting the maximum number of worker threads to one. So we thought we always had a situation where message order could be preserved.

Our initial testing with this design went well and it was also well-received in practice. Unfortunately it turned out that most users at that time used only very mild concurrency or had low message rates. With v4, this seemed to change. We now got reports on bad performance, and even performance loss when adding cores (and thus threads). Again David Lang provided us with very good evidence of what went wrong and also provided ideas for improvement.

We saw that the engine had a lock contention problem. Once this was clear, the reason was also quickly found: in a sense, processing was too quick. Receiving messages did not take much time. Each message received was enqueued. The output side, for many of the demanding environments, was also rather quick: a limited set of filters and some file write actions. The output side also dequeued messages one by one. In order to enqueue and dequeue messages the thread in question must acquire the queue lock[23]. The pthreads implementation on Linux utilizes very fast logic that works in user space, only, when the lock can be acquired. However, it needs to fall back to the kernel if a thread must be blocked. If we had a single input and a single worker thread, our design worked fairly well, as the worker usually finished execution quickly enough so that the input could acquire the lock without blocking. If the system became more busy, contention increased, but only to a limited level (as only two threads were involved). However, when more workers were added, lock contention dramatically increased and the time to process locking operations by far dominated overall processing time. This then lead to decreasing speed as threads were added. Our initial testing didn't spot this issue as we were primarily concerned with large, performance-intense filter sets and performance intense actions. So we simply missed this problem: we had simply forgotten the border case of very small, very fast filters and actions. And this unfortunately turned out to be a major use case for large scale deployments.

An obvious solution to this problem was to partition the workload. David Lang suggested to obtain a number of messages at once (called a "batch") while dequeueing. The core idea was that messages had no dependency on each other and so it would not really matter if we dequeued single or multiple messages. The obvious advantage of batching was that for the whole dequeue operation we only needed to acquire the lock once, resulting in far less lock contention. It is important to note here that this idea conflicts with the strict sequence requirement of messages. If we obtain batches, much larger reordering will happen than with single messages.

The next question to ask was why we did put all messages into just a single main message queue.

[23] This was a typical producer-consumer implementation

If we have multiple inputs, and we assume messages have no dependency on each other, then why not submit them to multiple queues, one per input? That way, the root cause of lock contention, at least at this level[24] were solved.

It turned out that our sequence requirement, and thus the traditional user perception of syslog, was the real root cause of our problem. So we analyzed that requirement and found it to be invalid. Actually sequence can only be counted on if there is no concurrency at all. So the only use case this is valid for is when we have a single-threaded process that originally emits messages, sends them to the syslogd (but no other process sends data to that syslogd) and that syslogd writes data to an action (but no other process writes data to that action). Also, the action must be sequence-preserving, because it must guarantee that messages can only be read in the same sequence they were written to the action. This is a property of sequential files and TCP streams, but not, for example, of UDP sockets. So, in short, we must have no concurrency at all inside the logging chain and logging going to a functionally restricted set of outputs, only. In all other cases, we cannot assume that we have a strict order of messages. To prove this, we need to look at different cases:

If the original log emitter is a multi-threaded process, it generates log data (or data that will lead to log data generation) on concurrent threads. Log data can only be emitted after a finite number of processing steps. The exact thread scheduling order cannot be predicted (think about events like external interrupts or page faults). As such, it is not predictable when the processing steps for a log entry will be completed. As such, which log entry is emitted first is depending on the scheduling order. For example, thread a may be one step in front of the final submission of his log message m_1 when it is preempted by thread b. Thread b logs message m_2 before context is switched back to a, which then does the final step and so logs m_1. Which message sequence is now right - (m_1, m_2) or (m_2, m_1)? There can be argument towards both cases, but the real answer is that we do not have any clear sequence indicator when looking at that level.

Next comes the transport from application to syslogd: some transports may not preserve order at all (UDP is an example). Even if they do, the context switch issue described above can reorder messages. So we cannot assume order is preserved if more than a single process writes to the syslogd.

The same problem applies to the syslogd. Again, context switches can, now severely, reorder messages. Just think about a TCP receiver thread a that runs concurrently to a local Unix sockets receiver thread b (a typical scenario). Let us assume both are busy. Now a initiates an OS call to read a 64K buffer. While doing so, b reads single messages from the local socket. Both read messages from buffers. We do not know which messages were actually received in front of which others in the buffers. Even if we knew, we would not know which message was sent earlier, because the first message may have been affected by a communication error and been retransmitted. At that point, the other message could have overtaken it. But even if we assume the sequence inside the buffer would be correct, the two receiver threads now run concurrently. For example, a's buffer will contain 800 messages (all received "at once" from a's point of view. Let us assume a uses up his time slice after having processed 500 messages. Now, b becomes active again and may read a message from the local socket that received after a returned from the read call (and thus definitely later than any message in a's buffer). This message is submitted to the queue by b, which then suspends itself. At this point a becomes active again and submits the remaining 300 messages from its buffer. Looking at queue order, we now have 500 messages from a, the single message from b and then 300 messages from a. Obviously, queue order does not reflect reception order. More importantly, queue order is almost random, or more precisely depending on scheduling of the input tasks. The very same problem can be seen when multiple workers write messages to outputs.

One extreme case is also worth noting: if the communication channel between a sender and a receiver is down, messages must be queued for later transmission. Let us assume the channel is down for one day. After the channel has been reestablished, all those messages are transmitted, and will most probably be intermixed by much newer messages the receiver receives from other senders via other channels.

[24] It is important to keep on one's mind that if messages end up in the same file, some form of synchronization is required sooner or later. But later is obviously much better, because much processing needs to be done before the message finally ends up in an action, if at all.

In conclusion, neither the original generator, the transports, nor the syslogd can ensure strict ordering of message sequence if at least one of them runs inside a concurrent environment. All modern systems and network protocols support various level of concurrency, so it is safe to assume that in almost all practical cases, the sequence in which messages are stored or emitted is not a proper indication of the order of events.

If order of messages is important, an order relation must be established. It is particularly hard to find a strict order relation that can be efficiently calculated across multiple systems. However, this often is not necessary. What is really needed is a strict order for events closely related to each other. Such events actually form a set that describes a higher-level event. Then, it usually is sufficient to find a precise enough partial order of these event sets. We will not describe that idea in detail here. However, it must be mentioned that a high precision timestamp usually is a very usable item to base orders on. It is not fully sufficient, though, because the precision is not good enough to differentiate events that quickly follow one after another. Also, there is the well-known problem of time synchronization across multiple machines.

These problems are not unknown to log analyzing tools. In practice, they are usually able to extract a sufficiently good order of events from the logs provided. This task will become simpler in the future: RFC5424 [5] provides both high-precision timestamps and a sequence indicator (RFC5424, section 7.3.1). These facilities can be used to provide excellent information on the order of events. One approach is to use Lamport clocks [8] to reliably provide a temporal partial order. Again, we can not elaborate on these details here.

The overall finding is that order of message reception is not reliable, and order of events must be established via some other way. Thus we can conclude that there is no value in trying to preserve as much of the reception order as possible at the syslogd level. And this means we do not need to restrict our use of concurrency. So, the problem we originally saw did not actually exist and we are free to implement any algorithm that fits within our goals.

This was a very important finding for the rsyslog project. It probably was the single most important fact that enabled us to provide a highly scalable engine. The thing to understand here is that we needed to break with traditional concepts in order to be able to provide much better service. To do so, we needed to think on a very fine-granular level, what then enabled us to prove that the traditional concept was simply wrong. At the same time, we identified the real problem (finding the order relation) and potential solutions (high precision timestamps, Lamport clocks).

Working on the basis that physical order of single messages is not important and order information must be kept within the message itself, there were no interdependencies between messages left. That meant we could process them in whatever order we wanted. This enabled us to partition with ease and select highly concurrent algorithms. The rest of this chapter will describe the optimizations in detail.

7.1 Workload Partitioning

If we can partition a workload, we do not need to do any synchronization at all for this part. So partitioning is obviously the best solution, and necessary to gain full speedup from multiple cores. Based on our findings, we were able to heavily apply partitioning.

7.1.1 Output Batching

We implemented David Lang's suggestion of batching. The queue consumer now dequeues all messages up to a configured maximum from the queue. The default batch size is 32, which sounds rather low but already means that we can reduce locking calls to $1/32^{\text{th}}$ on a busy system. We have experimented with batch sizes of up to 1024 messages. Our experience is that the extra speedup is very mild. For an extreme high-end system, this may make a small difference, but the default of 32 seems to be very good for many cases[25].

[25] A notable exception is database outputs supporting transactional mode. There, larger batches may scale very well because of the decreased overhead of putting many messages into a single transaction.

7.1.2 Input Batching

We applied a similar concept to inputs. Some of them are able to receive multiple messages at once. They now can create a batch of to-be-submitted messages locally and submit the whole batch at once to the queue. This dramatically reduces lock contention, and together with output batching it now is uncommon to have two threads compete for queue locks (on a busy system). There were also some less obvious gains from input batching: the queue engine needs to manage some data structures for each element enqueued (for consistency or for managing the worker thread pool). Without loss of functionality, we could reduce these management functions from once-per-message to once-per-batch. Depending on the configuration, this can be a very valuable saving. As such, we use large default input batch sizes (around 1024 messages, depending on input).

7.1.3 Multiple Main Queues

We implemented the ability to use multiple main queues and bind specific inputs to specific queues. For obvious reasons, this means that no lock contention can ever happen between these inputs. That functionality is especially useful if different inputs are being directed to different sets of outputs. In this case, there is unlimited concurrency between these two "processing units". It should be noted that this could also be achieved by simply running multiple instances of rsyslog at the same time. However, this involved some additional overhead and was not very popular with users. Now, we can gain the same performance by running a properly configured single instance.

7.2 Improving Locking

There are still a number of places inside rsyslog code where synchronization is required (for example in the case where multiple threads access the very same message object concurrently, a case that can happen). During the initial tuning effort in 2009, we did only minimal optimization to this code. Still, it proved to be rather useful. As a side note, in the spring 2010 effort we did some more work on locking and expect to do some major reworking in the third effort, currently scheduled for the winter 2010/2011 time frame.

7.2.1 Atomic Operations

We were able to replace some locks with atomic instructions, most notably atomic increment and decrement. They provided a trivial level of lock-freedom and work quite well for the cases where we could apply it. As this obviously requires hardware-specific features, it hurts portability. Initially, we just added an abstraction layer (via C preprocessor macros) but did not provide an implementation for anything but gcc-supported systems. Not unexpectedly, this caused troubles for some environments. Later, we added a generic replacement functionality where the operations are made atomic by guarding them by a mutex. This does not provide the same performance metrics, but at least permits rsyslog to run on any platform, no matter if atomic instructions are supported or not. It must be noted that almost all current systems provide atomic instructions, it is just a question of the driver level to properly utilize them. We plan to make more heavy use of atomic instructions in our planned third tuning effort.

7.2.2 Simplified Locking Primitives

At some places of the code, we used recursive mutexes. These are slower than non-recursive mutexes. So we redesigned the code in a way that can work with non-recursive mutexes. We were also able to fine-tune some parameters in respect to detached threads and cancelation modes. This brought some performance benefit, and also lead to cleaner and easier to read code.

7.3 Move Code to Places with higher Concurrency

We reviewed where in the message processing pipeline functionality resided. Most importantly, messages were parsed in the input, and only after that submitted to the queue. The input runs on a single thread, so any activity done there cannot be done concurrently. So we moved over message parsing from the input part to the first step of queue worker processing. In essence, this means the pipeline logically remained the same, but we went parallel in an earlier stage.

We changed that early in the tuning effort. Interestingly, some users actually saw performance worse than before the change. The reason was that we had increased lock contention, because the input now had even less to do, and so tried to acquire the queue lock even faster. That was solved by the partitioning work, and then the benefit of the pipeline modification became visible. This is a good example of an optimization that sounds good, but may work out really bad due to the overall design. Such an incident probably alerts on a design problem.

As a side-note, moving the parser code out of the input stage also helped to properly structure the parser subsystem. That in turn was a great aid to later improvements done.

7.4 Reduce Hidden Locks

As already elaborated in section 6, the malloc subsystem keeps at least a free list and some management information. Obviously, malloc calls from multiple threads must be synchronized in some way. So malloc does not only have the properties discussed in section 6, but it also has some implicit locking associated with it. If frequently called, malloc can considerably add to the locking overhead.

As part of our effort we reduced the number of malloc calls, which aided both in respect to memory access as well as locking. It should be noted that some other libraries may also potentially do some locking. Besides the performance effect, this also must be carefully locked at in order to prevent deadlocks.

8 Conclusion

The performance tuning work done in 2009 brought rsyslog's message processing rate up from 40k message per second (mps) to roughly 250k, as reported by some users. This is an impressive speedup factor of more than six. It must be noted, though, that part of the speedup is the improved support for multiple cores, so rsyslog can now process more messages, because it can consume more system resources. As multi-core machines are now standard in high demanding environments, we did not care to measure performance for single core systems. However, the speedup there is more a factor of two, at best close to three. This immediately draws the conclusion that the concurrency-related optimizations were the most important ones.

Nevertheless, we are not yet satisfied with rsyslog's multi core performance: it scales very far from linear[26] and there is much room for improvement. In spring of 2010, we began the second tuning effort, this time primarily directed at increasing the scalability by improving concurrency. During that effort, we heavily applied refactoring and introduced the idea of lock-freedom. We also redesigned a number of algorithms to explore existing concurrency in a better way. While this effort was only planned as an interim step, early results show a further speedup of four. In theory, this means that on sufficiently-equipped machines, rsyslog should now be able to hit the one million mps mark before no future improvement is possible. We have *not* yet received any feedback from the field that this is actually possible and wait for some of our power users to adopt the new code base.

We have planned a third stage tuning effort for the winter 2010/2011 timeframe. Then, we will focus on lock-freedom. Preliminary work looks very promising and we expect another notable speedup. The goal is to finally fully explore the independentness of messages from each other. Our hope is that this third stage will finally provide near-linear scalability based on the number of cores, effectively eliminating the upper limit of what a single rsyslog instance can process.

[26] We have a speedup of around 70% for the first cores added, with numbers rapidly decreasing if many cores are added

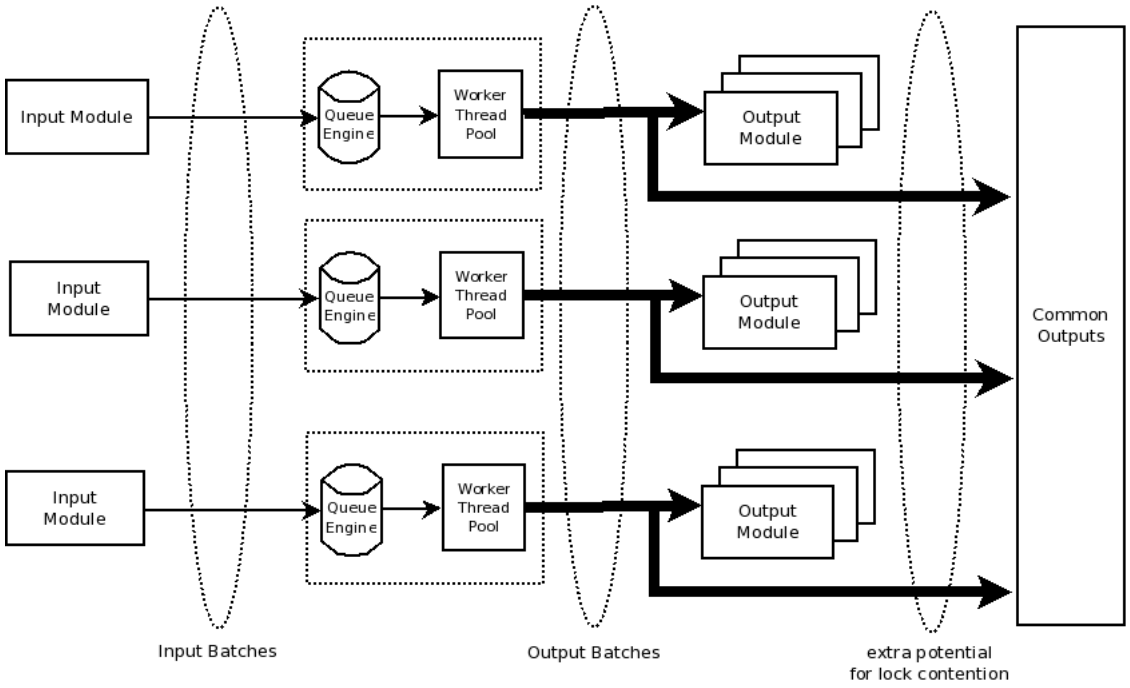

Figure 5: (Very) rough overview of rsyslog design after optimization: Idealized, all inputs can now send data to their own "main" message queue, each of which has its own worker thread pool dispensing messages to independent outputs as well as shared ones. All messages flow in batches of many, dramatically reducing locking overhead. In the not so ideal world, at least some inputs still deliver messages to single main queues, but batching still removes lock contention to a sufficient degree.

All tuning measures contributed to the overall improvement. The traditional optimizations had probably the least effect, whereas refactoring and especially the memory-subsystem related changes made a big difference. But by far the strongest effect was contributed by a real breakthrough: It was the finding that we had to break with traditional perception of how syslog works. As long as we worked on the basis that message sequence must be "preserved", we were unable to explore the full power of modern hardware. This requirement, by its very nature, demanded strong serialization and thus prevented us from going fully parallel. Once we knew preserving sequence was not even possible, we were able to utilize highly concurrent algorithms. Only this enabled us to reach new magnitudes of processing speed. It is rather educating to compare the high-level design overviews given in figure 2 (before the tuning effort) and figure 5 (after it). The enhanced concurrency can very clearly seen in figure 5.

So in our point of view, re-evaluating current practice and questioning old habits is probably a key ingredient of moving from the mostly sequential programming paradigm to the fully concurrent one demanded by current and future hardware.

Acknowledgments

Rsyslog is a team effort and we would like to thank all those active users that provided feedback and advise on the rsyslog mailing list or are otherwise involved with enhancing rsyslog. Without their continuous work, we would not have been able to achieve the presented results. We would also like to express our sincere appreciation to all those that shared performance metrics with us. Without that data, we would never have been able to know which processing rate rsyslog can reach on high-end equipment. Last, but not least, we would like to express special thanks to David Lang, who most notable introduced the idea of batching, and some person we unfortunately can not name due to NDA, who provided the

idea of multiple main queues. Both were key ingredients for the initial performance tuning effort.

References

[1] The linux-kernel mailing list faq. http://www.tux.org/lkml/#s15-3. [last accessed 2010-08-04].

[2] Serge Demeyer. Submitted to icse'2003 maintainability versus performance: What's the effect of introducing polymorphism?, 2002.

[3] Ulrich Drepper. What every programmer should know about memory, 2007.

[4] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[5] R. Gerhards. The Syslog Protocol. RFC 5424 (Proposed Standard), March 2009.

[6] Rainer Gerhards. rsyslog project public git. http://git.adiscon.com/?p=rsyslog.git;a=summary.

[7] Poul-Henning Kamp. You're doing it wrong. *Communications of the ACM*, 53(7):55–59, 2010.

[8] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[9] C. Lonvick. The bsd syslog protocol, 2001.