

Locks, Deadlocks and Abstractions

Experiences with Multi-Threaded Programming at CloudFlare, Inc.

Ian Pye

CloudFlare, Inc.
ian@cloudflare.com

At CloudFlare[1, 2], we are about a year into our public release. Over the last six months we've seen exponential growth. CloudFlare provides a content delivery network currently serving over ten billion page views/month to over 200 million unique visitors. During July 2011 approximately ten percent of all people on the Internet visited a CloudFlare powered site at least once. Figure 1 shows monthly page views served by CloudFlare over the past year.

We run a highly customized software stack on a limited number of powerful physical servers deployed in twelve data centers on three continents. The upshot of all of this is that we've been forced to rapidly code, and re-code, to take full advantage of 24 plus cores per machine.

This experience report is a very brief survey of the programming models and debugging methodology CloudFlare uses. We first describe two ways in which CloudFlare deals with concurrency issues. We then compare bugs and features in two applications which are representative of the above paradigms.

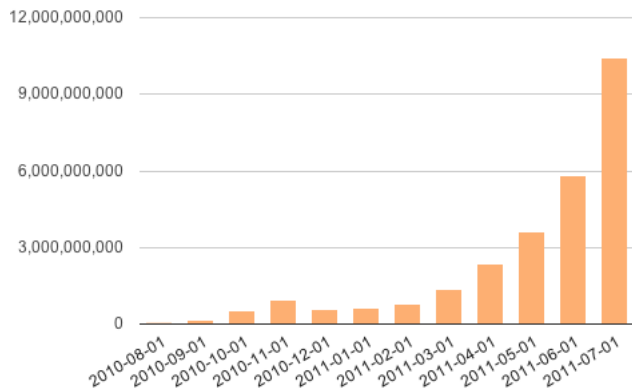


Figure 1. Page Views shown across the CloudFlare network by month.

Frequently, we write multithreaded code without knowing it. Two examples of these are the programs Ningx (C language), a web server and reverse proxy, and our DNS server (C++) implementation. Both of these provide a well thought out module system. CloudFlare has worked extensively extending both of the

above projects. Writing a module involves providing implementations of callback functions. The encapsulating framework provides that each implementing function can be written without regard to thread safety. Of course, any static data is the responsibility of the implementer to protect across differing thread's accesses, using semaphores, monitors or other synchronization construct. This works very well as long as there is a well designed framework which does what you need.

Sometimes though, this is lacking. In this case, we have had to develop concurrent implementations from scratch. One such system is CloudFlare's logging program. The next section is a case study of the various iterations CloudFlare's logging system has been through.

1. Logging From Prototype to Production

CloudFlare is essentially a giant proxy service for Internet applications. As such, we generate a large amount of web access logs. The logging system needs to monitor these logs, providing aggregate graphs on demand for a small number of fixed queries. See Figure 2 for sample output. In practice, this boils down to incrementing a lot of counters. For example, the site linuxdating.com received ten HTTP GET requests in the period from 12:00 to 12:15. The challenge lies in building a scalable system which can deal with Terabytes of data ingested per day.

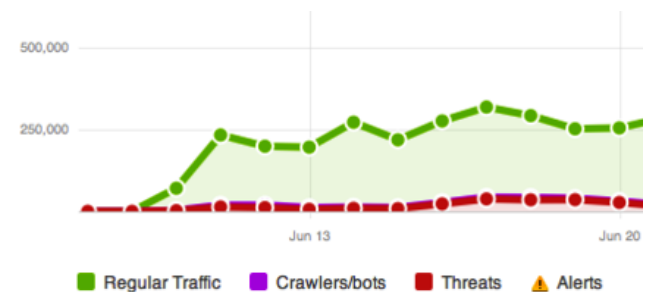


Figure 2. Sample output from the logging program. Page Views by hour categorized by type of viewer (Human/Bot/Threat). See cloudflare.com for more detail on these categories

Built originally without live data, the first prototype succumbed to a sea of buzzwords. Here, a Clojure script processes files, writing them into a HBase[9] database. Clojure provides some great concurrency constructs, including futures, parallel map over a list and watchers[3]. Unfortunately, it also runs on the JVM. In practice we found that this implementation, even when debugged, required a large amount of memory to perform well. It also exhibited a memory leak which proved very difficult to track down. For reasons which are outside the scope of this paper we decided to not

continue with HBase. Looking to simplify the deployed machine images we also decided to do away with all Java dependencies.

Falling back, we re-implemented a prototype log processor in perl. The heavy lifting of data storage is done via an embedded DB. The database used, Kyoto Cabinet (KC)[5], supports transactions which threads obey. Threads outside of a transaction ignore transaction locks. But, the call to `begin_transaction()` will block until the calling thread has exclusive access. The DB also supports opportunistic locking where instead of blocking the DB will return a null value when it cannot acquire a lock. In practice though we do not use this feature. A concurrently model where multiple threads all interact with a single DB instance, with critical sections being controlled via transactions in the DB suggests itself. This has worked well for all subsequent iterations. The DB further provides an atomic increment function, which since all we are doing is incrementing counters works very well. This limits the need for multi-operation transactions.

The single threaded perl implementation was fast to implement. It worked well but was soon falling behind under peak load. Taking sixteen minutes to process fifteen minutes of logs is not a sustainable process. Re-writing this into C++ gained some time but it was clear that a multi-threaded version would have to be produced. Kyoto Cabinet has Map/Reduce functionality built into it. This could be very useful going forward. At the time however, this feature was still being prototyped and did not support parallel execution of map or reduce jobs. Jobs were instead executed sequentially in the calling thread. Current versions of KC do now allow for an arbitrary number of mappers and reducers to be executed in parallel.

Instead, we fell back on the Boost threading library[4] to count lines in parallel. We choose Boost over Posix Threads or another threading library because we were already using other Boost libraries and have greater experience with Boost threads. Primarily we use scoped locks (which unlock automatically when they go out of scope) as well as the thread group construct. The process is to create variable set of threads, each called on the same function with differing arguments. These all process a section of the input file in parallel. The spawning thread waits for all threads in the group to join and then writes out an output file. This model allows us to experiment, cranking up the number of concurrent threads until CPU overhead and disk IO start decreasing performance. In practice, this follows a parabolic curve, peaking at a maximum speedup with 16 threads on a 24 core machine. Above 16, performance starts to actually decrease.

We find that the `atop`[10] command is most useful when needing to quickly evaluate performance on a real system. This provides a per-core measurement of CPU utilization, along with per-volume IO usage. Optimizing becomes a game of making the most number of CPU's be red (high utilization), while trying not to hammer the filesystem too much. Figure 3 shows some sample output. Of course, having an all-red display leads to game over for your server.

2. When Things Go Wrong

Debugging strategy is largely failure driven. A developer is informed by an angry ops team that updates are not happening as they should or a process in production has experienced a segfault. Or, we notice a particular process is not performantive enough to keep up with CloudFlare's projected growth. Obvious bugs are largely squashed by now. Instead, obscure inputs (CloudFlare is web facing so we have no control whatsoever over these) will trigger a failure with some (usually very low) degree of probability. Typically, reproducing the inputs will not lead to being able to re-produce the failure deterministically. Instead, we'll either run code repeatedly until an error is observed or, where practical, radically bump up the number of parallel operations until the flaw occurs frequently enough to facilitate debugging.

ATOP - 16log1		2011/08/14 21:00:37			
PRC	sys 1.29s	user 16.38s	#proc 424		
CPU	sys 38%	user 547%	irq 9%		
cpu	sys 1%	user 99%	irq 0%		
cpu	sys 0%	user 100%	irq 0%		
cpu	sys 1%	user 99%	irq 0%		
cpu	sys 1%	user 99%	irq 0%		
cpu	sys 1%	user 99%	irq 0%		
cpu	sys 11%	user 30%	irq 4%		
cpu	sys 5%	user 0%	irq 0%		
cpu	sys 1%	user 0%	irq 0%		
cpu	sys 9%	user 15%	irq 1%		
cpu	sys 4%	user 0%	irq 2%		
cpu	sys 0%	user 0%	irq 0%		
cpu	sys 0%	user 0%	irq 0%		
cpu	sys 0%	user 0%	irq 0%		
cpu	sys 0%	user 0%	irq 0%		
cpu	sys 3%	user 0%	irq 0%		
cpu	sys 1%	user 0%	irq 0%		
cpu	sys 0%	user 1%	irq 1%		
CPL	avg1 1.60	avg5 0.78	avg15 1.03		
MEM	tot 47.3G	free 132.5M	cache 37.7G		
SWP	tot 20.0G	free 19.9G			
PAG	scan 48171	stall 0			
DSK	cciss/c0d0	busy 58%	read 0		
DSK	cciss/c0d1	busy 53%	read 0		

Figure 3. Arcade style optimizing. Blue is good, red is better – except when there is too much of it. Output of the `atop` program.

`Gdb`[11] is the tool for debugging segfaults, using a strategy of running until an error is encountered and then examining the stack trace. We do not currently perform any static code analysis. We will run Valgrind[6] (a dynamic analysis tool) on occasion though. We use Valgrind for both performance profiling and also the Helgrind[7] race-detector tool. Why Valgrind? Because there's reasonable documentation on it online and installing Valgrind on Debian Linux is as easy as `aptitude install valgrind`. We would certainly be open to a useful static analysis tool, if one is both easily installed and well documented.

Once an error is not observed any more for enough iterations and the output after the fix is the same as before the fix (for our collection of test inputs), we consider code correct. This is the real beauty of programming for the web – pushing updated code is dead easy so we can afford to be somewhat non-rigorous software engineers.

We now present a list of bugs found in two relatively small projects. The first is a custom backend to our DNS system. It is a primarily single threaded collection of callback functions. There is also a singleton class which provides a wrapper around two embedded databases. Two threads in this class monitor the filesystem, taking input from Linux's `inotify`[8] system whenever a file changes. On a file change, the thread will start a transaction, update an in-memory database, and then commit the transaction. The callback functions read from the same databases. All writes are done inside of a transaction. In practice, this system can handle over 12,000 queries per seconds while still allowing for the dynamic updating of DNS records.

The second is the logging system described above. This is a green-field piece of code where all threading functionality is implemented by CloudFlare. Table 1 showcases the size, parallelism constructs and bugs found in each project. Parallelism constructs include database driven transactions and traditional locks. Bugs are classified as either a race condition or deadlock.

3. Lessons Learned

All of the bugs found in DNS stemmed from the custom change notification system, which is outside of the callback structure. The most damaging bug was a deadlock which prevented updates from being noticed by the database. However, since reads were still

Table 1. A comparison of two multithreaded software projects at CloudFlare.

	<i>Logger</i>	<i>DNS</i>
Model	Native	Callback
LoC (Excluding Frameworks)	2,833	1,276
Source Commits	85	79
Transactions	5	6
Locks	1	0
Maximum Concurrent Threads	16	6
Race Conditions	8	3
Deadlocks	0	2

working fine, it took quite a while to realize that something was wrong. Debugging the problem took even longer. The deadlock was a classic case of two threads each trying to acquire the same two locks. One thread in development and initial deployment ran on a small input list. This resulted in a very quick execution time and reduced the potential for a deadly embrace. As soon as input size was increased beyond certain size, failure struck.

For the *Logger*, the bugs are all centered around synchronizing writes. Since the databases are shared across all threads, the priority for performance is limiting the sections of code which are inside of a transaction. Unfortunately, this leads to over-optimism on how small a critical section can be. Typically these types of race conditions only will exhibit themselves with large inputs and high concurrency numbers. For this reason, we find it key to test in development with the same concurrently values as in production or higher, even though since we are developing on a two core virtual machine this leads to significant performance degradation while testing.

Acknowledgments

Thank you to all my co-workers at CloudFlare, especially Lee Holloway. I also thank my shepherd, Benedict Gaster.

References

- [1] <http://www.cloudflare.com>
- [2] <http://www.crunchbase.com/company/cloudflare>
- [3] http://clojure.org/concurrent_programming
- [4] <http://www.boost.org/doc/html/thread.html>
- [5] <http://fallabs.com/kyotocabinet>
- [6] <http://valgrind.org>
- [7] <http://valgrind.org/docs/manual/hg-manual.html>
- [8] <http://linux.die.net/man/7/inotify>
- [9] <http://hbase.apache.org/>. HBase is a distributed hash table a-la Google's BigTable.
- [10] <http://linux.die.net/man/1/atop>
- [11] <http://www.manpagez.com/man/1/gdb/>