Final Firewall Report

By Graham Carling and Benjamin Koatz

Table of Contents

Synopsis	1
How to Run	3
Analysis of Functions that use Concurrent Data Structures	4
Comparative Results from Cached vs. Non-Cached Runs	6
Comparative Results from Serial vs. Parallel Runs	7
Histograms	8

Synopsis

In general we were able to implement most of what we set out to do in our Interim report. We were able to provide an intelligent locking structure on our concurrent data structures and set up a multithreaded framework that works and works quickly. There were a few interesting design decisions we had to make, notably around our histogram lock, caching, and our implementation of R. We will discuss all of this below.

At a high level, our design involves the use of 2 thread pools. One is very large and used for data packets, and the other is small and used for config packets. This is because config packets are effectively processed serially, so there is little use in having more than 2 threads work on config packets. So our data packet thread pool has 254 threads and our config packet thread pool has 2 threads, which meets the requirement that there are a maximum of 256 packets being processed at any given time.

Our program works by first running the initialization phase and putting a number of configuration packets - the number of addresses exponentiated by 1.5 - into the system. We waited for all these packets to get completely processed. Then we create random data and config packets and feed them into a method that puts them in the proper pool. This happens on loop for some number of times, and we measure our execution time using this standard number of packets, calculating how much time it takes for all of them to be fully processed.

As we said above, config threads were effectively run serially - since they wrote to our concurrent data structures, and as we will discuss below write-lock lock-striping produces errors, the only way to effectively guarantee proper behavior was to have a general lock for config threads. Data threads, however, since they were mostly reading (and sometimes writing to our histogram), did not have a general lock, and relied on locks around specific data structures when needed.

Our design uses 3 main data structures: PNG, R and histogram. We use a variety of locks to ensure that these data structures remain consistent. The locking strategy is as follows:

For starters, we use lock striping on PNG and R. We use 100 locks for each, so for a given index i (the address that is the key in this case) we lock the lock numbered i % 100. This gives us a nice way to ensure that when locks are taken the whole data structure is not locked up. In addition, we use read write locks on these data structures, further freeing things up so that multiple data packets could be reading information from the same index at the same time .

For our histogram, we initially had a similar scheme involving lock striping. However, we ran into an issue - concurrent writes to our data structure lead to failures in our hash map on slower computers. At the root of these failures was concurrent writing to different locations on a hash map. Unlike PNG and R, which because of the nature of our config threads were never written to concurrently, the histogram was initially being written to. Once we isolated this as the cause of our problem, we replaced stripe locks on that map with a coarse-grained lock on the entire data structure.

When we process config packets we write lock the appropriate parts of PNG and R at the start of packet processing. This is because config packets must be processed atomically and we cannot have a situation in which some data packet thread sees an old value in PNG and the new value in R: the change must occur all at once. Then the config packet processor makes the appropriate updates to PNG and R and unlocks these locks.

When we process data packets, we read lock PNG, check the target address against the information there, then unlock PNG; then we read lock R, check our target address, and then unlock R. If it all works out, we lock and update our histogram and then unlock it.

This whole situation used to be slightly more complicated when we had a cache. We have removed the cache (though the relevant code remains commented out in our files), because caching slowed down our program immensely (timings are discussed below). Here’s the way our cache was implemented:

For the permissions cache we used a single lock. This is not ideal, but we found that it was impossible to do the type of iterative cache invalidation that we wanted to do while only taking out a lock on part of the data structure. The issue was that any form of iteration over the structure would result in a ConcurrentModificationException if another thread made an update to

the cache while this was happening. So we had to lock over the full cache to make sure that another thread couldn't modify it while we were doing the cache invalidation.

When we processed data packets, we first locked the cache lock and checked if the target address pair was in the cache. If it was, we used it. Else, we would get the new permission and update the cache with this value. At the end of this process, we would check where in the packet train we were, and if this packet was at the end of a train then we would dump the entry for this address pair from the cache to save space, and re-add it later if the same pair showed up in a different train.

The coarse-grained locking and necessary updates slowed down our implementation more than the instantaneous access sped it up. Thus in our final version we decided to remove the caching from our algorithm, though we left the code and descriptions in to show how we reached this conclusion.

Finally, our last relevant implementation detail is how we implemented R. R is a map from one destination address to the set of source addresses which can send packets to it. This set of source addresses is stored in two ways in our backend - as a list of acceptable address ranges and as a set of acceptable addresses. The list of ranges is easier to store than the set, since it is shorter, but the set is easier to check. Checking if an address is in a list of ranges is O(n) on the size of the list, while checking set membership is O(1). However, if, for example, an address accepts messages from all possible sources, it would be silly to store all addresses individually rather than a range of them.

That's why we swap the two types depending on the nature of our source list. We start out using the range list. If that range list gets longer than ⅕ of the addresses, we switch to a set implementation, since the benefit of using the range list lessens. If our set then accepts from more than ⅘ of the addresses, we switch to a range, since at worst that can result in n/5 different ranges (if the non-present addresses are spaced evenly), which is the limit we previously outlined above. After switching we clear out the previously full set/list. The AddressHandler class performs all this silently in the background, with generic insertAddressRange, removeAddressRange and containsAddress methods that access and fill the right data structure. Though this isn't exactly related to concurrent programming skills, we thought it was a nifty addition to the optimization of this project.

<u>How to Run</u>

Run the main file in PacketFilter.java. You can alter the parameters as you wish: Use values 1-8 on 'num' on line 97 to cycle through pre-set values easily.

Analysis of Functions that use Concurrent Data Structures

Since our functions are not claiming to be lock- or wait-free, we will prove that every function which accesses a concurrent data structure (PNG, R and the hisogram) is deadlock- and starvation-free:

1. PacketFilter's construction and filterPacket methods
   a. Deadlock-Free and Starvation-Free (and Lock-free and Wait-free)
      These methods are deadlock and starvation free since they either instantiate the data structures, or they simply pass the concurrent data structures to be handled by new threads, which is just transferring reference values and not actually doing any reading or writing. These method will never be blocked by any locks. In fact, these methods are both lock and wait free.

2. DataThread's AccessAllowed method
   In this method, there are two sections of interest. We first lock and utilize PNG, then we lock and utilize R. Since these do not overlap, i.e. we can deal with these instances separately:
   a. Deadlock-Free
      There are only two ways in which this method could deadlock: either at the PNG lock or R lock. Two threads running this code cannot deadlock with one another because they both read lock the same lock, so they won't have to contend with one another. So the only way this could deadlock is with a config thread running its run method. I can discuss here why this won't happen, and in doing so also explain why the config run method works.

      In the config run, we first lock the PNG lock, then the R lock for a given range. Clearly if the ranges don't overlap we don't risk deadlock, so let's assume the addresses we are working with will stripe to the same values. If a config lock holds the PNG lock, then they will also necessarily beat the data thread to the R lock, meaning that the config thread fully finishes, then the data thread can go. If the data thread gets the PNG lock first, then the config thread has to wait. Once the data thread is done with the PNG lock, it is possible that the config thread gets both the PNG and R lock, which is fine, as it then finishes. Otherwise the data thread gets the R lock, finishes, and then the config thread finishes. In any order, it is impossible for a deadlock to occur here, and these locks are the only way a deadlock could occur. This means that this method is deadlock free.
   b. Starvation-Free

      We know we won't deadlock, so the only concern is that a single thread never gets to go. This won't happen because we are using the Java ReentranLock which we know gives us starvation-free locking, so this method is starvation free.

3. <u>DataThread’s Run method</u>
    a. Deadlock-Free
       This method locks in only 2 places: one is in a call to accessAllowed, which we addressed above. The only other place it locks is at the histogram at the bottom. This is just a single lock that only one thread can be in the critical section of at a time, which very trivially cannot deadlock.
    b. Starvation-Free
       Since we know we won’t deadlock, for the same reason as the above method we know that this is starvation free

4. <u>ConfigThread’s Run method</u>
    a. Deadlock-Free
       As discussed above, this method is deadlock free. It cannot contest with another config thread because this method effectively runs serially, and it won’t deadlock with data threads as we already showed.
    b. Starvation-Free
       Again, this method is starvation free for the same reasons given above.

Comparative Results from Cached vs. Non-Cached Runs

Here we have some times in nanoseconds to run our code on 100,000 packets (after initialization config packets). We used all 8 sets of run parameters specified in the handout.

Run Parameters	Cache Run Time (ns)	Non-Cache Run Time (ns)
1	1,060,746,210	317,693,702
2	463,310,243	292,224,956
3	550,549,568	289,617,151
4	581,396,176	118,139,981
5	1,330,654,994	194,810,741
6	2,585,181,775	122,959,543
7	1,271,738,099	142,360,939
8	5,313,062,068	220,181,657

We can see pretty clearly from the above table that the added overhead of locking our cache actually caused our program to run significantly slower. Because of this, we made the decision to remove the cache - the handout talks about this issue, and asks when it makes sense to cache with the complications that it brings. We decided that was not worth it, so the table below reflects the serial version compared against our code without the cache, so our fastest runs.

Comparative Results from Serial vs. Parallel Runs

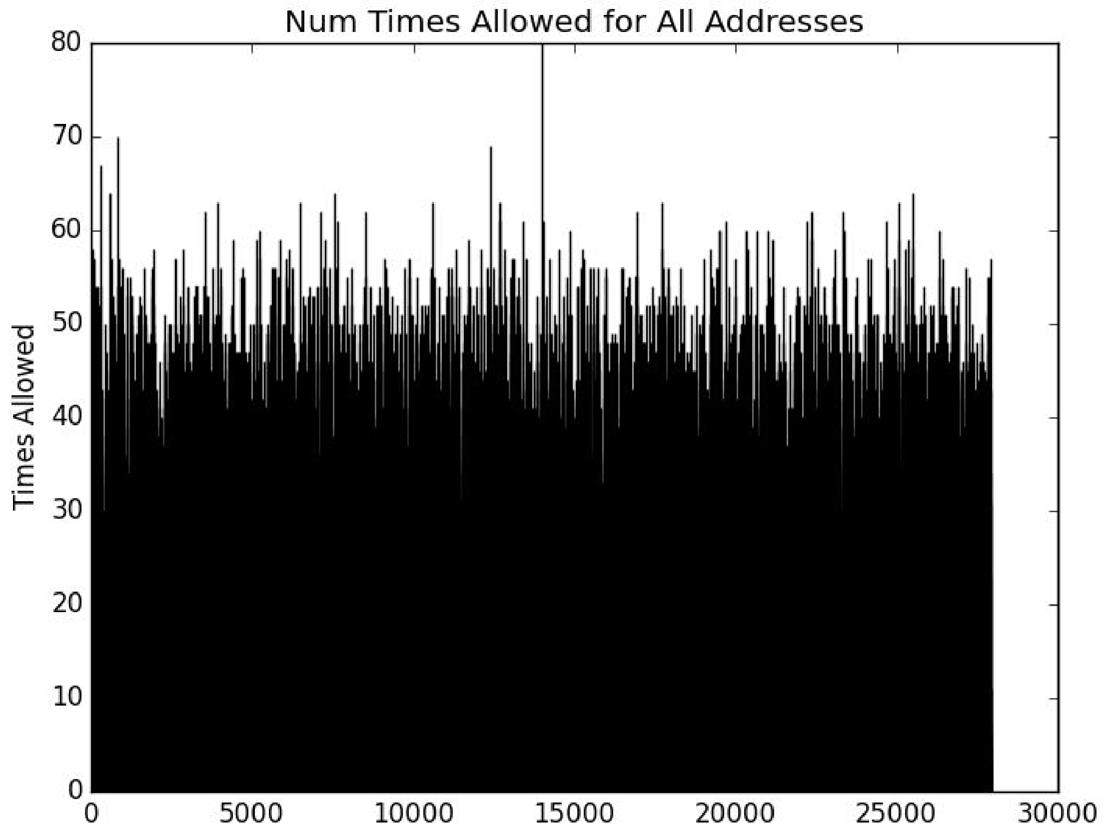
Run Parameters	Serial Run Time (ns)	Parallel Run Time (ns)
1	1,060,746,210	317,693,702
2	463,310,243	292,224,956
3	550,549,568	289,617,151
4	581,396,176	118,139,981

After the 4th run our serial version took so long that we decided it wasn't worth bothering to run all of them and include them. We think the biggest bottleneck in our firewall is processing configuration packets - we get the best speedup in parameter group 4 which has the lowest config packet ratio of the options listed above. It's also possible that other runs would perform differently - these were all taken from single runs but the speed of the parallel version could change a decent amount between runs.

Histograms

Here are example histograms, using the following parameters and with 500,000 data packets being sent after the initial configuration step:

numAddresses Log	numTrains Log	meanTrain Size	meanTrainsPer Comm	meanWindow	meanCommsPer Address	meanWork	config Fraction	png Fraction	accepting Fraction
11	12	5	1	3	3	3822	.24	.04	.96



This is the histogram for all addresses. Times Allowed are the number of times that address was allowed to successfully submit a data packet:

And, for a more fine-tuned look, here's just the first 100 addresses: