This is a term project description from the MIT version of this course. Your job is to figure out how to turn it into a capstone project. You should pick a subset of these features to implement, and clear them with the instructor. You should turn in a detailed design, due November 5, 2015, which details the overall architecture of your design. Your final report, due December 10, 2015, should present your final design, code, and performance numbers. Specifically, for every concurrent data structure you should explain what it does and why it is correct.

1 Project Overview

We will be building a firewall from scratch: we will see a stream of *packets* from various source addresses (represented as arbitrary integers, not necessarily in a compact range) and perform several operations before sending them to the destination address (also integers) specified in the header of the packet. By default, packets will be *data* packets, messages from source address to destination address with some payload. Our firewall will perform two primary functions: 1) enforce access controls and 2) filter the payloads for evidence of *bad guys* (that the packet is from a suspicious source).

Access control: A primary function of the firewall is to filter packets based on a set of permissions and calculate a checksum of the packets that are permitted to pass while dropping those that are not. These permissions restrict which data packets may proceed based on their source and destination addresses and will occasionally be modified by *configuration* packets. There are two types of permissions:

1. **There are permissions that dictate which addresses (in this assignment, an address is a 4-byte `int`) are permitted to send packets. We will refer to this rule abstractly by the name PNG, where $PNG[S]$ is a `boolean` specifying whether a source address, S, is permitted to send packets, irrespective of the recipient.**
2. **There are also permissions that dictate which sets of addresses a *particular* destination address is willing to accept packets from. We will refer to this rule abstractly by the name R, where $R[D]$ is the set of source addresses from which a destination address, D, is willing to accept packets.**

Filtering: Packets which make it through the access control pipeline will be filtered to check for potential file-sharing. In particular, we will measure a checksum on the body of each packet and histogram the resulting values. The rationale is that the buckets in the histogram corresponding to often-shared files will be inordinately large and traffickers in such data payloads are potential bad guys. We will not actually take action based on the contents of the histogram, however. We are merely collecting statistics that will be used by some other hypothetical software to make changes to the permissions via configuration packets. Also, in a real system, we would deliver the packets that make it through the access control pipeline to output wires. In our system, once you update the histogram (or drop the packet if it fails the permissions check) then you are free to discard it.

The Assignment

This assignment will proceed in two phases:

- **System Architecture: In this phase you will begin by building a serial reference design of the firewall system.**

  This phase will culminate in an interim report, due November 5, 2015, which details the overall architecture of your design. The interim report should demonstrate that your design satisfies the specifications and explain the rationale for your design

choices. In addition, you should hypothesize about the primary limitations to performance of the lock-based design in the next phase and what approaches you will pursue to overcome them.

- **High-Performance Design: Next, you should evolve your System Architecture from the previous phase into a high-performance firewall using concepts from the lectures, previous assignments, and your impeccable engineering intuition. Your report, due December 10, 2015, should present your final design, showing how it adheres to the specifications and is correct. Specifically, for every concurrent data structure you should show your reasoning for why each method is linearizable and whether it is deadlock-free, starvation-free, lock-free or wait-free. You should detail your exploration of the design space, describing your hypotheses about performance effects and the experiments you design to test those hypotheses. Finally, you should characterize the scalability of your design (*eg.* throughput as a function of number of threads divided by the throughput of the serial version) and the primary mechanisms you use to achieve that scalability.**

The Specifications

You are given a class called `PacketGenerator`, which can not be concurrently accessed by multiple threads and can not be modified. This class represents a network packet stream with a format which is out of your control. In a real system, packets arrive on the wire and there is a somewhat complex back-pressure mechanism to negotiate the rate at which the packets arrive. In our system, we are going to avoid the back-pressure complexity and instead you will just *pull* the packets via the `getPacket()` method (again, this is a serial method) and thus consume packets at precisely the rate you process them. However, you will need to guarantee that there are never more than 256 packets in flight at any time, which mirrors the real-time nature of a real firewall; delivering low-latency performance in a firewall system implies bounded state due to Little's Law.

There are two types of packets (of type `Packet`) returned by `getPacket()`, *configuration* packets and *data* packets, which are distinguished by an `enum`, `MessageType`, with corresponding values `ConfigPacket` and `DataPacket`. The class `Packet` also includes pointers to the classes `Config`, `Header` and `Body`.

- **Data Packets: (`Packet.type=DataPacket`) Data packets carry a payload called a `Body`, which has two fields, `iterations` and `seed`. To process a data packet means to apply the `Fingerprint()` method (from previous assignments) to the those fields. This is in contrast to a *real* packet body, which would just be a long piece of memory containing data. The `Fingerprint()` serves as a proxy for calculating a checksum on the body of a packet, which would ordinarily involve streaming the long piece of memory through the core. We will use the fingerprint as a way of detecting commonly transferred (*ie.* identical) bodies, allowing us to detect potential file sharing. Point-to-point file sharing protocols typically break up long (media) files into smaller chunks, a *packet train*, each piece of which would be identical to the same corresponding chunk shared by another person, thus generating the same fingerprint as well. You should create a histogram of the fingerprints, which take values in $[0, 2^{16})$, of the bodies of packets which are permitted to be processed according to the rules below (*ie.* a data packet from source address S to destination address D is permitted to be processed only if $PNG[S]$=`false` and $S \in R[D]$). It is essential that you do not calculate `Fingerprint()` for packets that are not permitted to be processed - this is the root of a *Denial of Service* attack - coercing the firewall to do unnecessary of work in order to slow down other traffic. Data packets also carry a `Header`, which has the following fields:**
  - **`source` - the source address.**
  - **`dest` - the destination address.**

- `trainSize` - the number of packets from `source` to `dest` making up the entire message between the two.
- `sequenceNumber` - the packets in a train are ordered by `sequenceNumber`, counting from 0 to `trainSize`-1.
- `tag` - a pseudo-random number used to distinguish two concurrent trains between the same `source` and `dest` addresses.

- **Configuration Packets:** (`Packet.type` = `ConfigPacket`) Configuration packets are used to modify the permissions of a particular address in two contexts: 1) when the address is a source address (*ie.* whether it is allowed to send packets) and 2) when the address is a destination address (*ie.* the set of source addresses it is willing to accept packets from). As before, we encapsulate the permissions with two theoretical data structures R and PNG. Recall that $R[D]$ is the set of source addresses from which a destination address, D, is willing to accept a packet. $PNG[S]$ is a `boolean` that specifies whether or not a source address, S, is allowed to send packets, irrespective of the recipient. You are free to implement the functionality of these two data structures however you like, but we will refer to them abstractly as R and PNG in the coming text for convenience.

  If two configuration packets are processed concurrently, they must be done so atomically - that is, it must appear as though they were processed serially, though there is no specific requirement on the particular serial order. You should demonstrate the serializability of configuration packets in the section on correctness in your report. Configuration packets populate the `Config` class (but not `Header` or `Body`), which has the following members:

  - `address` - the update is applied to this address (an arbitrary `int`).
  - `personaNonGrata` - this `boolean` specifies whether `address` is permitted to send packets (`false`) or not (`true`). The operations sets PNG[`address`] =`personaNonGrata`.
  - `addressBegin` and `addressEnd` - this half open range of addresses [`addressBegin`,`addressEnd`) will either be included in or excluded from (depending on the setting of `acceptingRange`, below) the list of source address intervals previously accepted by `address`.
  - `acceptingRange` - this `boolean` specifies whether the range of source addresses [`addressBegin`,`addressEnd`) should be accepted (`true`) or rejected (`false`) by destination address `address`, overriding any previous permissions to that range of source addresses.
    - Example 1: if destination address 9 previously accepted packets from source addresses $R[9]$ ={2-4, 5-7} and received a configuration packet with `acceptingRange` = `false` and address range [4,6), the new range of acceptable source addresses for destination address 9 would be $R[9]$ ={2-3,6-7}.
    - Example 2: if destination address 5 previously accepted packets from source addresses $R[5]$ ={1-2,4-5,7-9} (indeed, the permissions even apply when sending packets to yourself!) and received a configuration packet with `acceptingRange` = `true` and address range [2,5), the new range of acceptable source addresses for destination address 5 would be $R[5]$ ={1-5,7-9}.
    - Note: a data packet from source address, S, to destination address, D, is permitted to be processed if $PNG[S]$=`false` and $S \in R[D]$. Further, this test must be serializable with all configuration packets. For example, it is not permitted for a configuration packet to modify $PNG[S]$ or $R[D]$ in the interim between a data packet (from source address S to destination address D) checking $PNG[S]$ and then checking $R[D]$.

The Packet Generator

- **Parameters - The `PacketGenerator` is instantiated with several parameters which all affect the nature of the traffic it generates:**
  - **`numAddressesLog` - the log (base 2) of the total number of addresses in the system.**
  - **`numTrainsLog` - the log (base 2) of the total number of active packet trains.**
  - **`meanTrainSize` - the average number of packets in a packet train.**
  - **`meanTrainsPerComm` - the average number of packet trains sent between two addresses in any particular *communication* (a sequence of back-to-back packet trains between the same addresses constitutes a communication).**
  - **`meanWindow` - the average number of addresses that are *active* at any given time. That is, when a new packet train is generated, the source and destination addresses are chosen from a restricted set of active addresses - `meanWindow` is the average size of this set.**
  - **`meanCommsPerAddress` - the average number of communications that an address is part of (as either source or destination address) while *active*. Of course, an address can (and will) become active again, but only after an extended period of inactivity.**
  - **`meanWork` - the average number of `iterations` in the body of a data packet.**
  - **`configFraction` - the fraction of packets which are configuration packets ($\in [0, 1]$).**
  - **`pngFraction` - the fraction of configuration packets with `personaNonGrata=true` ($\in [0, 1]$).**
  - **`acceptingFraction` - the fraction of configuration packets with `acceptingRange=true` ($\in [0, 1]$).**
- **Initialization - After instantiating the `PacketGenerator` with the parameters above, you should process $A^{\frac{3}{2}}$ (where A is the number of addresses in the system or $2^{\texttt{numAddressesLog}}$) configuration packets (using `getConfigPacket()`) before starting the worker threads and measuring throughput. This will ensure that the permissions tables are in steady state.**

Design Considerations

- **In order to process configuration packets atomically, it will be necessary to take out locks on multiple data structures simultaneously. If all of your data structures are amenable to *lock striping* (as in PSET4 or Figure 13.6 in the text), you could consider consolidating the locks into a single bank yielding potentially better cache performance. Tread carefully; protocols taking out multiple locks are notoriously prone to deadlock, so consider yours very carefully. Be sure to justify why your locking protocol is deadlock-free in the report.**
- **While configuration packets and the permissions lookups for data packets need to be processed atomically (updating the histogram of `Fingerprint()` results need not occur atomically with the permissions lookups, however), this does not imply that any of your underlying data structures need to be linearizable or anything stronger than deadlock-free. That said, you should explain what properties your data structures possess, providing the reasoning for why in the report. However, if you feel that you could provide the atomicity constraints of the firewall specifications while using potentially simpler data-structures, that is a perfectly valid approach.**
- **A nominal implementation of R (where $R[D]$ is the set of source addresses from which a destination address, D, is willing to accept packets) is a hash table (indexed by**

destination address) of skip lists (organizing a tree of address intervals). This type of data structure would require $O(\log n)$ time to do a lookup, but fails to take advantage of the observation that packets between particular source / destination address pairs are repeated over the length of a train and even over multiple trains (comprising a communication). In order to exploit this observation, you might consider caching the result of a permissions lookup for a source / destination address pair. When we refer to a *permissions lookup* or the caching of such permissions, we just mean whatever state your design requires in order to adhere to the specifications above. For instance, a cached copy could be as simple as a `boolean` saying whether a data packet from source address, S, to destination address, D, is permitted to be processed or not. Whatever the details of your system, here are some general considerations regarding caching:

- **Keeping track of cached entries - Cached copies of permissions lookups for a source / destination address pair may be the only state that a thread sees when deciding whether a packet may be processed, thus they must be kept coherent with the *real* state, R and PNG. For instance, if $PNG[S]$ transitions from `false` to `true` due to a configuration packet for `address`=S, then all cache copies with source address S must be removed (or otherwise reflect the new state). Or, if a configuration packet with `address`=D, `acceptingRange`=`false` and address range $[S_0, S_n)$ were processed, all pairs $\langle S_i, D\rangle$ $\forall i \in [0, n)$ must be removed from the cache. So, there is a question of how you go about finding the cached copies that must be removed in these scenarios.**
- **Revoking cached entries - In particular, if a configuration packet changes the permissions such that a cached copy of the permissions of a particular source / destination address pair is no longer valid, it must not persist after the updates to R and PNG. This is because processing configuration and data packets must be linearizable with respect to one another. The most straightforward way to do this might be to lock down the entries corresponding to $R[D]$, $PNG[S]$ and all cached copies derived from them - that's a lot of locks! Because the cache is (presumably) indexed by a hash of both source and destination address, you would need to be careful in the order you acquire locks to avoid deadlock.**
- **Optimistic deleting - Cached copies should provide exactly the same result that one could obtain by looking up R and PNG directly, so the cached copies can be invalidated prior to processing the configuration packet with no change to correctness. So, one could consider an optimistic strategy where one serially deletes all of the cached copies that a configuration packet would require, knowing that another thread could simultaneously add some cached copies that conflict. However, in the typical case, no other thread *will* actually add any additional copies and then when you take out locks to process the configuration packet atomically, you have less total work to do (potentially cleaning up any copies that were added in the interim) and thus less lock contention.**
- **Pruning - Eventually, you will see every possible pair of addresses as a source / destination address pair. So, if you want to avoid the situation where your cache grows to be the square of the number of addresses (hint: you do), you will need to prune your cache. Data packets have the fields `sequenceNumber` and `trainSize`, so you could know which packets are the end of a packet train (though, it is possible for slight reordering between packets in a train, since you can have 256 in flight at a time...). Do you prune an entry at the end of a packet train, knowing that a packet train between the same addresses may follow? If not, how would you know to come back and prune it later?**
- **To cache or not to cache - So, with all of these complications ... under what circumstances does it even make sense to cache?**

- **In past assignments we have made use of a very simple worker model; we had one `Dispatcher` thread and n `Worker` threads, where all `Worker` threads did the exact same thing. Of course, there are many other ways of partitioning work among threads with many tradeoffs in overhead, cache behavior and scalability. Here are some examples:**
  - **Pipeline - You could choose to implement a pipeline style parallelization where threads are assigned to tasks and packets will be partially processed as they pass through each pipeline. For instance, you could have a thread pool which only looks up permissions in the cache and queues up packets which are permitted to be processed for another thread pool to calculate the `Fingerprint()`. More complex operations, like configuration packets or data packets which miss in the cache (*ie.* their source / destination address pair is not in the cache), could be handed off to yet another pool of threads which handle these special cases, potentially with simpler synchronization - that is, you might try to arrange it so that a single thread handles all packets which could possibly need to take out multiple overlapping locks. Of course, how many threads should be mapped to each of these categories? How do they depend on the parameters of the `PacketGenerator`? Pipelines also incur more copying overhead between stages - how would you decide whether the work in each stage justifies the cost of copying?**
  - ***Partial* Pipeline - The trickiest part of the firewall application is handling configuration packets atomically - could you create just two pipeline stages, handling configuration packets and data packets with separate thread pools? If you could serialize the configuration packet processing stage, it may simplify synchronization. Then again, how many data packets per configuration packet would you need for this to not be the bottleneck?**
  - **Worker Model - Each worker processes a packet, whatever type it happens to be, to completion. It is alluringly simple, but what are your expectations of cache behavior and lock contention?**

  Whatever worker model you use, you should consider the requirement that to be scalable and maintainable for future hardware systems, you should be able to get more performance by adding more threads. What is your scaling strategy for when we move beyond the 6.S193 machine in the hypothetical future?

Experiment

You should characterize the performance of your parallel firewall on the following set of parameter mixes. That is, for each mix, instantiate the `PacketGenerator` class and test the throughput as a function of n (number of worker threads) divided by the throughput of the serial version. Discuss in your report how the different corners of this parameter space affect performance and what inefficiencies, if any, they expose in your design.

	Parameter Mixes							
Parameter	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
`numAddressesLog`	**11**	**12**	**12**	**14**	**15**	**15**	**15**	**16**
`numTrainsLog`	**12**	**10**	**10**	**10**	**14**	**15**	**15**	**14**
`meanTrainSize`	**5**	**1**	**4**	**5**	**9**	**9**	**10**	**15**
`meanTrainsPerComm`	**1**	**3**	**3**	**5**	**16**	**10**	**13**	**12**
`meanWindow`	**3**	**3**	**6**	**6**	**7**	**9**	**8**	**9**
`meanCommsPerAddress`	**3**	**1**	**2**	**2**	**10**	**9**	**10**	**5**
`meanWork`	**3822**	**2644**	**1304**	**315**	**4007**	**7125**	**5328**	**8840**
`configFraction`	**0.24**	**0.11**	**0.10**	**0.08**	**0.02**	**0.01**	**0.04**	**0.04**
`pngFraction`	**0.04**	**0.09**	**0.03**	**0.05**	**0.10**	**0.20**	**0.18**	**0.19**
`acceptingFraction`	**0.96**	**0.92**	**0.90**	**0.90**	**0.84**	**0.77**	**0.80**	**0.76**

This seemingly random smattering of parameter configurations mirrors a familiar scenario in the industry, where you might be confronted with various application spaces, each with their own peculiarities. It is rather challenging to design something that performs well in all scenarios, so instead we sample the scenarios we care about and use those parameters to reign in our designs. This holds for processor design as well as many large software projects.

BONUS: There is a ninth set of parameters which is vaguely in the space of the eight above. You should specify how your code should be configured (all parameters, including number of threads) to maximize the throughput of this mystery workload. The two best lock-based implementations (highest throughput for 5000ms after the initialization) will receive an additional 5% on their final grade.

Writeup

Please submit typeset reports summarizing your results from the experiments and the conclusions you draw from them,. Each of these reports should be a standalone document that would explain to a reader unfamiliar with this material and course what you did and why. You should also submit a *.jar* file of your development directory, containing all of the working code.