Designing and Implementing a Multithreaded, Synchronous Packet Processing Firewall

Authors: Cesar Guerrero, Conor Wuertz

Abstract

This firewall enforces access controls and monitors packet payloads for evidence of malicious packets. It uses multiple threads to speed up the processing of packets and is carefully designed to ensure processing of packets and modification of access controls are serializable.

How to Compile & Run:
We have included runnable jar file exported from the maven project we created.

run program:

```
java -jar Firewall.jar numAddressesLog numTrainsLog meanTrainSize
meanTrainsPerComm meanWindow meanCommsPerAddress meanWork
configFraction pngFraction acceptingFraction parallelOrNot
```

for parallel version, set parallelOrNot to 1, for serial set it to 0.

We have included our java files in a zip file called source. You can look in com/Firewall/app/ to find our source java code.

Pipeline & Thread Pools

Fixed sized thread pools are created by invoking the newFixedThreadPool factory method in java.util.concurrent.Executors. This approach was compared to using a CachedThreadPool that can support a non-fixed number of threads (instead invoking newCachedThreadPool). The CachedThreadPool approach sometimes caused OutOfMemoryExceptions and provided no boost to the runtime.

MainThread:

- *Initializes all data structures and components. Takes care of initial processing to bring configurations to steady state. Then pulls packets from the PacketGenerator and delegetates them to the thread pools.*

Deque & Categorize Thread Pool:

- *This pool dequeues packets from the packet queue, checks the type of the packet, and then calls ConfigPool.queuePacketToProcess(packet) or DataPool.queuePacketToProcess(packet), which sends the packet to the ConfigPool or the DataPool.*

Thread Pool For Handling Config Packets (ConfigPool):

- *queuePacketToProcess(Packet ConfigPacket) will add a packet the pool's task queue (task queue provided and handled by executor)*

Thread Pool For Handling Data Packets (DataPool):

- *queuePacketToProcess(Packet DataPacket) will add a packet the pool's task queue (task queue provided and handled by executor)*

The total number of threads allocated to the pools is 256 because this is the limit on the number of packets that can be in flight. The number allocated to the Config pool is TOTAL_CONFIG = (int)(TOTAL_THREADS * configFraction * CONFIG_HANDICAP), where configFraction is an input from the user, and CONFIG_HANDICAP was a constant that we experimented with. Although our thinking was that Config tasks might pile up in the config pool as they take more time to complete, CONFIG_HANDICAPS greater than 1.5 actually slowed our runtime, so we left it at 1.5. Then the number of threads allocated to the Data pool was imply the remaining number of threads, TOTAL_THREADS- TOTAL_CONFIG.

Runnables (Tasks for threads in thread pools)

ProcessConfigPacket(Packet ConfigPacket)

- *updates Permissions (details in the **Atomic Modification of Permissions** sections)*

ProcessDataPacket(Packet DataPacket)

- *checks permissions*
- *calculates Fingerprint()*
- *writes to Histogram*

Implementation of Histogram

Fingerprints are unique when packet bodies are distinct. Counting occurrences of unique fingerprints allows us to to monitor potential file sharing.

The Histogram will be represented programmatically as a ConcurrentHashMap<Fingerprint, Occurrences>

Packet storage:

To guarantee that there are never more than 256 packets in flight at any time, we are using the atomic integer *packetsInFlight* to keep track of how many packets are in flight. It will be incremented by threads that dequeue from the packet queue and decremented when packets have been successfully processed by the pipeline.

Before the main thread calls getPacket(), it firsts check that FlightCount is not 256. If it does, it waits and checks again in a while loop.

Implementation of PNG[S]

The abstract PNG rule will be represented concretely in our program as a ConcurrentHashMap<S,Boolean>. This will map source adresses S to boolean values that state whether or not S is permitted to send packets. We argue that no cache is required for these lookups because looking up in a HashMap has an expected runtime on the order of O(1). Java's ConcurrentHashMap is a good fit for our application as "even though all operations are thread-safe, retrieval operations do *not* entail locking". Furthermore, "Retrievals reflect the results of the most recently *completed* update operations holding upon their onset" . Additionally, to ensure the best possible performance from our ConcurrentHashMap, we will construct it using the initialCapacity and concurrencyLevel arguments. The initialCapacity argument will be used to size the map optimally for the number of addresses we expect-- $2^{NUMADRESSESLOG}$. The concurrencyLevel indicates the number of threads that will concurrently modify the map and is used by the ConcurrentHashMap to partition the map in such a way as to maximize the number of threads that can (lock-striping). We will set it to the number of threads that are allocated to Config Packet handling.

Implementation of R[D]

The abstract R[D] rule will represented concretely as a ConcurrentHashMap<D,TreeRangeSet<S>. That is, destination addresses D will be mapped to tree range sets of permitted sources S.. The ConcurrentHashMap was chosen for the same reasons mentioned in ***Implementation of PNG[S]***. However, R[D] must map D to a set and not to a boolean value, so its ConcurrentHashMap maps to a TreeRangeSet<S>. The documentation states that this TreeRangeSet is based off of a TreeMap, so get, put, and remove all run in guaranteed O(log(n)) time.

There is a tradeoff here of memory vs performance. The optimal performance could be provided by mapping D to a HashSet of permitted sources. This would provide O(1) lookup but could grow to a quadratic size if destinations have very broad ranges of addresses in their R[D]. We choose to use a TreeRangeSet to minimize memory usage at the cost of settling for O(log(n)) operations.

Atomic Modification of Permissions:

Our initial solution for the atomic modification of permissions (please see our Final Capstone Project: Detailed Design document) ensured only that threads dedicated to the processing of Config Packets would perform their modifications atomically.

The most difficult portion of the assignment was ensuring that the permissions lookup is serializable with all configuration packets. For example, it is not permitted for a configuration packet to modify P NG[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 crux of the problem is that even with ConcurrentHashMaps that can be modified or read atomically, modifying *both* in one atomic step requires protection. Our solution is to use two arrays of ReentrantReadWriteLocks(), one for PNG and one for R. In order to avoid creating an enormous number of locks (i.e. one for each address), we mod the address by a stripeFactor. This implementation gave us greater flexibility in experimenting with how “fine-grained” our locks would be. After careful analysis, it was set equal to

Before checking for permissions, our data threads get the PNG read locks for their source, and the R read locks for their destination. By using read/write locks, multiple readers can lookup permissions concurrently. Upon finishing their check, they release the locks.

Before modifying permissions, our config threads get the PNG and R write locks for their address. They then make their changes and release their locks.

This protocol is not lock-free or wait-free, but it is linearizable and approximately starvation-free.

It is linearizable because every method that modifies or reads the permissions for its addresses does so only after acquiring corresponding locks for R and PNG . We can show this via contradiction. Suppose that the thread A is reading the permissions for S and D. Let us also suppose that the corresponding values in either R or PNG were modified after thread A acquired the appropriate read locks. But no thread can modify R or PNG at these particular addresses unless it holds the write lock for these addresses, and it is impossible for another thread to hold the write lock if thread A holds the read lock. Thus, it is actually impossible for R and PNG to be modified at these addresses while thread A reads them. Similarly, if thread A is holds the write lock instead and modifies the permissions, no other thread can be holding a corresponding read lock or write lock, and thus no other thread is reading or modifying those particular permissions concurrently.

This protocol is also approximately starvation free due to the fairness option set on the ReadWrite locks. This ensures that threads contend for entry using an approximate arrival-order policy (from the ReentrantReadWriteLock documentation). Because of this arrival-order policy, threads will eventually receive locks, and thus eventually complete

Serial Comparison Implementation:

When the last argument to the program is not 1, the firewall runs serially. This is in order to provide a baseline reference implementation. It uses the same logic and the same data structures but without the locking and thread pools.

Performance Data and Discussion:

As the fraction of packets that are config packets decrease, our parallel throughput generally increases. This is expected, as config packets are more time and resource consuming to process. In addition, taking out write locks to process them means that data packets can't check permissions concurrently. This pattern is also evident in the sequential implementation, which also speeds up as the fraction of config packets decreases. As the meanWork variable grows, the amount of time taken for a Fingerprint() to be processed increases, and thus the throughput decreases. This is also reflected in the results of our Firewall.