An Approach to detect DDoS attack

Context:

- DoS (Denial of Service) is a malicious attempt to affect the availability of a targeted system, a website or an application, to legitimate end users.
- Typically, attackers generate large volume of packets or requests ultimately overwhelming the targeted system.
- In DDoS, the attacker uses multiple compromised or controlled sources to generate the attack.

Problem:

- A customer runs a website and periodically attacked by a botnet in a DDoS attack. You'll be given a log file in Apache Log format from given attack. Use this log to build a simple real-time detector of DDoS attacks.

Approach

- What can I offer to this problem?
    - A system that can detect bot-IPs (near)real-time, enables you to take control of your traffic without causing latency or downtime to your users.
    - A light weight, object oriented, multi-threaded application that can be dynamically configured to handle variety of DDoS scenarios.
    - A system that can be deployed in a variety of host environments without the worry of incompatibility.
    - A system that can be scaled to handle larger traffic volumes as it grows locally or globally.
    - A system that integrates seamlessly with your server/application systems with clearly laid out contracts (interfaces).

- How do we differentiate botnet traffic from normal traffic?

    Constraints:
    - DDoS attacks can happen across Infrastructure layers (Network, Transport) or Application layers (Presentation, Application).
    - Traffic patterns fluctuate based on many factors – shopping events, festivals etc. DDoS attackers can easily mask alongside high-volume traffic making it harder to detect.
    - Also, high number of bots with small request traffic / bot can overwhelm service when they coordinate and attack in high volumes. This system cannot detect such attacks.

My Approach:

- Let's assume the website can handle a maximum of 'X' requests per second. (This number could be determined by stress testing tools. For our purposes: 830 req/sec)
- This application will alert if the traffic level exceeds 'X' requests per second
- (And) A high level of "503 Service Unavailable" status codes

- How is real-time detection achieved and what is the frequency of bot detection?
  - Used Kafka cluster to publish and consume log messages. Offers high throughput with parallelism built-in.
  - Used time series Rolling Window to encapsulate messages for accurate view of traffic vs time.
  - Processing messages (for bot detection) is quick since it is done in-memory.
  - Flagged IPs (if any) are detected every 10 seconds. (configurable)

- Advantages Kafka offered?
  - Faster writes and reads
  - APIs (Producer and Consumer APIs) integrate well with Java.
  - Persistence (configurable retention period)
  - Easy to configure and test scenarios (Consumer groups)
  - Built-in parallelism.
  - Easy setup and deploy
  - Scalability and Fault-tolerance.

System Specifics – Implementation Details

Apache Log Ingestor (https://github.com/Yagnadat-Talakola/ApacheLogIngestor**)**

- Data Ingestion
  - Read messages from log file -> parse it to JSON -> write to *apache-log-msg* Kafka topic partitions.

- Message body

```
{
    "ip_address":"194.70.52.190",
    "request":"GET / HTTP/1.0",
    "bytes_sent":"3557",
    "browser":"Opera/9.00 (Windows NT 5.1; U; en)",
    "response_status_code":"200",
    "timestamp":"25/May/2015:23:11:52"
}
```

- How did I ensure faster and efficient writes to Kafka?
  - Making use of built-in parallelism in Kafka using partitions.

- send () invoked asynchronously to send messages to Kafka partitions. Instead of waiting for acknowledgement for each message (blocking), let that be handled by a separate I/O thread.
- acks: 1
  - fastest acknowledgement without waiting for in-sync replicas (we don't have in this project)
- Kafka Producer maintains a buffer for each partition. Efficient buffering ensures optimal send calls.
  - *Batch.size* determines the size of buffer
  - *Buffer.memory* determines total buffer memory available.
  - *(adds latency) Linger.ms* waits for the duration specified for the buffer to fill up before executing send ()

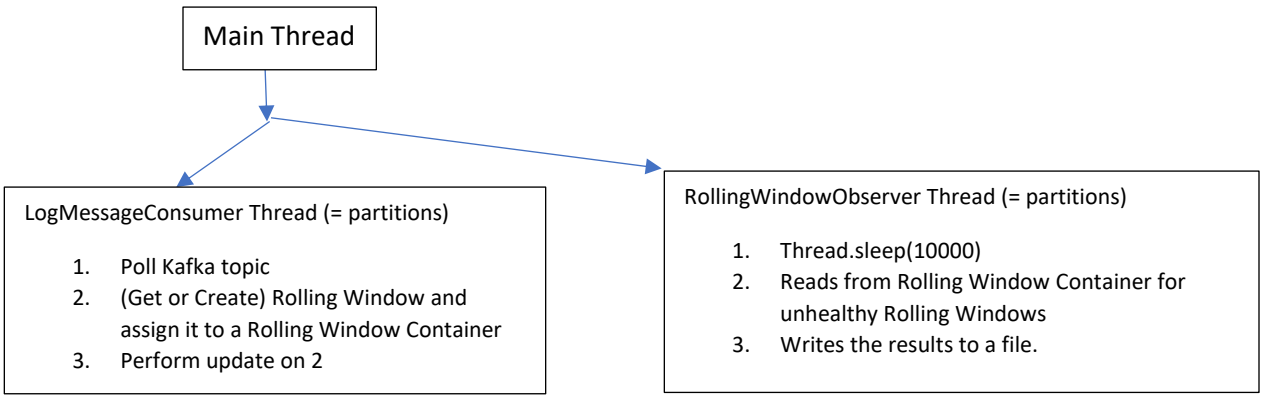
- Key implementation details:
  - IP address is set as the key to each message (record) written to Kafka. This ensures message(s) from same IP are routed to the same partition. This will help consumption easier to manage and process.
  - Record: {key: IP_address, value: JSON (message)}

Log Analyzer DDoS (https://github.com/Yagnadat-Talakola/LogAnalyzerDDOS**)**

- Core functionality: Read messages from Kafka topic -> Process messages and apply known DDoS patterns to flag IPs -> write the result to a file.

- Key implementation details:

  Process flow:



Rolling Window Container:

- Each instance of Rolling Windows Container is shared between LogMessageConsumer Thread and RollingWindowObserver Thread.
- Has a ConcurrentHashMap of form {key: LocalDateTime, value: Rolling Window} that stores Rolling Windows.

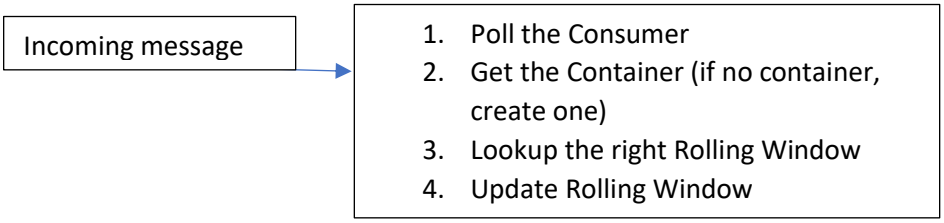
Container ID: 1
RollingWindowContainerMap

Concurrent Hash Map is thread safe (multiple threads can operate on a single object without any complications) Not possible for Hash Map. No locks on reads therefore read times are faster than a Synchronized Hash Map (which has locks on reads)

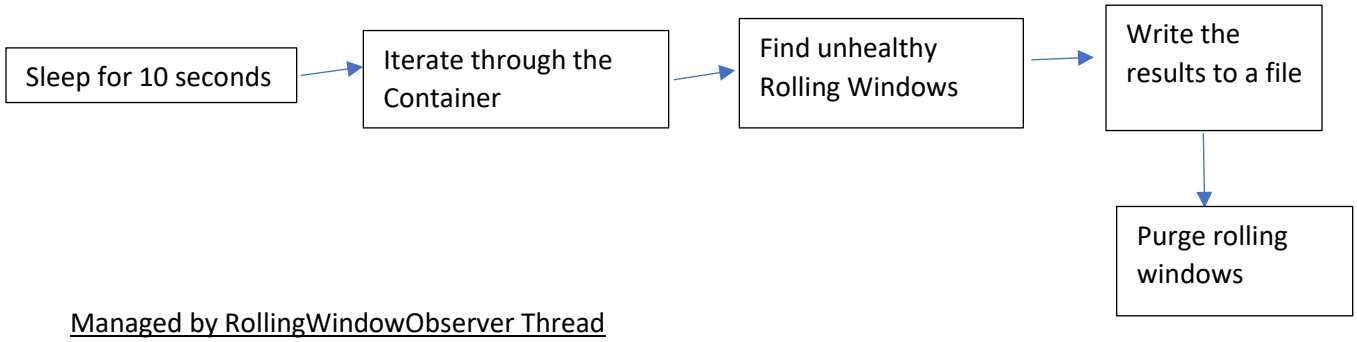
Rolling Window:

- Inbound messages are grouped in a Rolling Window of configured duration (30 sec).
- Each message is routed to an appropriate Rolling Window based on its timestamp.

Container ID
StartTime
EndTime
IPAddressCount Map
StatusCodeCount Map
TotalRequestCount



Managed by LogMessageConsumer Thread



Managed by RollingWindowObserver Thread

Rolling Window Operations:

- Create Rolling Windows: @params (message.timestamp, RollingWindowContainer)
    - Invoked when there is no Rolling Window which can fit the message (message.timestamp is not with in startTime and endTime of Rolling Window)
    - It creates (RETENTION_PERIOD / DURATION) Rolling Windows.
    - Store the created Rolling Windows in the Rolling Window Container.

- Get Rolling Windows: @params (ApacheLogEntry, RollingWindowContainer)
    - Iterates through the Container to see if there is a Rolling Window which can accommodate the message. (startTime < message.timestamp < endTime)
    - If yes, returns the Rolling Window.
    - If no, it invokes Create Rolling Window and returns the Rolling Window.

- Purge Rolling Windows: @params (RollingWindowContainer)
    - Why purging?
        - Free up space in the Container.
        - Purge those Rolling Windows whose startTime is outside of RETENTION_PERIOD.

System Limitations

- The solution presented above can only detect certain DDoS patterns where the request threshold is breached. There might be other DDoS patterns that may not be detected.
- Did not provide data/metrics on space and time utilization vs input load, throughput, processing capacities etc.
- Used a single Kafka broker & Zookeeper running on local docker containers. For production, it is better to use multiple brokers with higher replicas across data centers.

Scope for further iterations

- Currently we have a reactive system for detecting DDoS. Ideally, we should have a proactive and predictive system in-place to monitor requests and respond even before they reach our servers.
- API Gateway and Proxy servers would be a good place to place DDoS detection systems. The requests can be filtered against known DDoS patterns and take appropriate action before reaching app servers.
- Geographical filtering will help when we know or expect less to no traffic from certain regions.
- Use Machine Learning models to monitor and detect bot behavior. Use reinforced learning algorithms to improve prediction accuracy.
- Timeseries database like Graphite will be ideal to persist and process message events. This integrates very well with Grafana dashboards for rich visual statistics.