Technical Disclosure Commons

Defensive Publications Series

December 14, 2018

# MACHINE-LEARNING OF LEADING INDICATORS FROM A PLURALITY OF SERVICES TO PERFORM MEMORY MANAGEMENT ON ANOTHER PLURALITY OF SERVICES WITH CORRELATING MEMORY USAGE PROFILE

Jayesh Wadikar

Pok Wong

Ajay Madhavan

Smruti Lele

# MACHINE-LEARNING OF LEADING INDICATORS FROM A PLURALITY OF SERVICES TO PERFORM MEMORY MANAGEMENT ON ANOTHER PLURALITY OF SERVICES WITH CORRELATING MEMORY USAGE PROFILE

AUTHORS:
Jayesh Wadikar
Pok Wong
Ajay Madhavan
Smruti Lele

## ABSTRACT

Presented herein are techniques for machine-learning of leading indicators from a plurality of services and the use of these leading indicators to perform memory management operations on another plurality of services with correlating memory usage profiles. The techniques presented herein include collecting, as leading indicators, service event logs from a plurality of services. Machine-learning is then used to cross-correlate the leading indicators with memory use pattern snapshots at subsequent times for another plurality of services to predict the optimal when, how much, and how for execution of memory garbage collection on the latter set of services.

## DETAILED DESCRIPTION

The techniques for machine-learning provided herein help to save memory and resources and are applicable to both on-premises solutions and managed cloud services. These techniques will also help in reducing latency on applications which would increase the turn-around time for customers and result in better performing. The described techniques will help in troubleshooting performance issues, which is very difficult in the current state of the art.

Network and computer information systems depend on memory to operate. Microservices architecture with thousands of services each run with its own memory allocation provisioned in a container. Container orchestration systems manage the memory used by the containers to evict containers based on their priority and memory consumption. Certain managed cloud services automate and fine-tune the network and computing elements for optimal performance, including monitoring and allocating memory to where it matters. Certain managed cloud services rely on the assurance of their own optimal memory allocation to operate the network.

State of art memory management algorithms use incremental reactive heuristics which iterate to evict containers until the consumption falls back to the threshold. They prioritize to evict the highest memory consumer (memory pressure) to bring down the total memory utilization in the least amount of iterations. This was found to result in an undesirable effect of evicting the containers that need the memory but keep the container that doesn't.

Garbage collection running at the right moment could avoid eviction but it takes time to run, and running it at the wrong moment either unnecessarily impedes performance, or failed to provide sufficient memory in the time of need. Prior art techniques for determining when to run focus on memory usage momentum (memory pressure), *e.g.*, monitor the memory usage to predict if it is going to grow and how to collect.

Predicting a quantity would go up based on momentum without considering any supporting reason behind is reactive and could incorrectly be the opposite. The state of the art solution manages memory based on the reference pattern of the memory to decide if garbage collection should be done for that memory (*e.*g., a variation of the classic Least Recently Used (LRU) algorithm). This solution, however, is again momentum based. Managing memory based on a processor service's own use also fails to consider the inter-services interactions in a microservice environment.

Techniques are presented herein for machine-learning of leading indicators from a plurality of services to perform memory management on another plurality of services with correlating memory usage profiles. An overview of the techniques provided may be seen with reference to Figure 1 below. The techniques include collecting, as leading indicators, service event logs from a plurality of services. Machine-learning is used to cross-correlate with the memory use pattern snapshots at subsequent times of another plurality of services to predict the optimal when, how much, and how to execute memory garbage collection on the latter set of services.

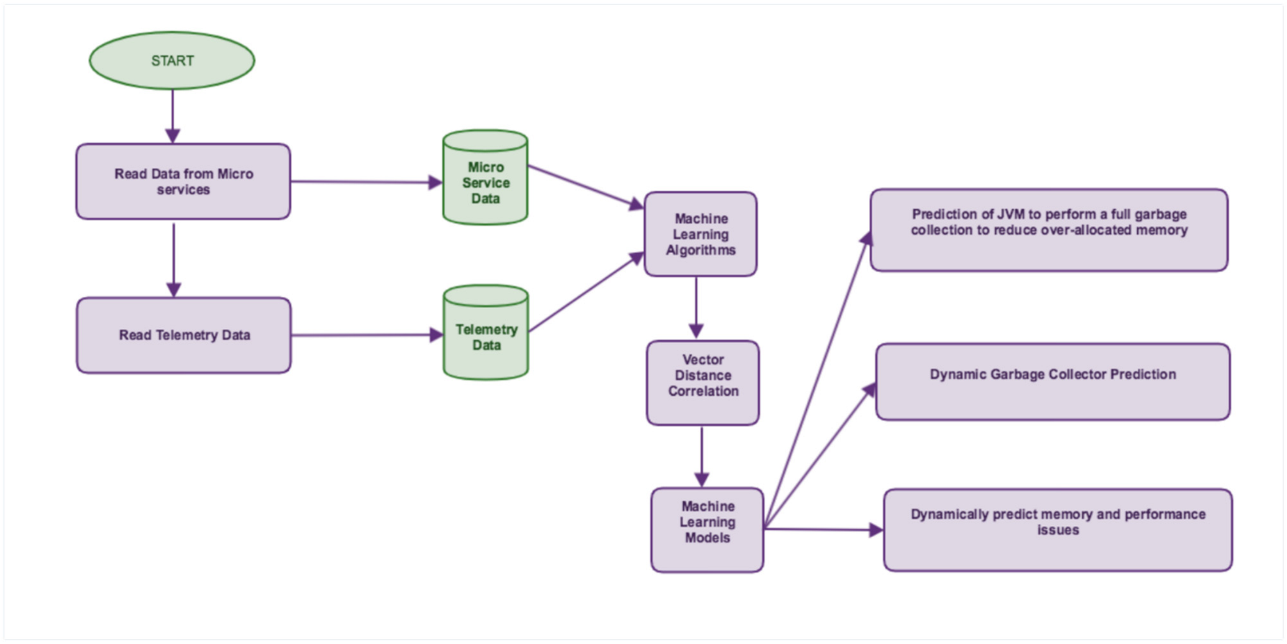
START
Read Data from Micro services
Read Telemetry Data
Micro Service Data
Telemetry Data
Machine Learning Algorithms
Vector Distance Correlation
Machine Learning Models
Prediction of JVM to perform a full garbage collection to reduce over-allocated memory
Dynamic Garbage Collector Prediction
Dynamically predict memory and performance issues

*Figure 1*

The described techniques use machine-learning to train and correlate service event logs to memory consumption. In contrast to the current state of the art methods, the described techniques are not based on examining the memory consumption or the event of one service at the time but all across and it looks for advanced indicators from training data. The rationale is that because service interaction outside of a service with a service could be an indicator of memory needed elsewhere moments later (*e.g.*, sending of a packet requires the receiver to have the memory before the receiving end received it and emitting the received event). The event logs from across all the services are used as the leading indicators that a particular service is going to need memory and for how long. Conversely, the event logs from all the services are also used as the leading indicator that a service is going to be relinquished and available for some other service in need.

Distance correlation is used to determine the event logs associated with a memory to be needed occurrence. Besides the detection of memory consumption, it is also important to detect if the garbage collection has sufficient time to run. When performing a full garbage collection, the application thread will be paused for some time. (*i.e.*, max pause garbage collection). A snapshot is taken to obtain the CPU idle time and correlate it to the event log that would signal the beginning and end of CPU idle period.

After a state of art classifier, such as word2vec, is used to normalize the event into categorical keyword encodings (*e.g.*, init, ready, send, receive, etc), the present techniques encode the log event with the producer of the event, and correlate it with the memory usage pattern of another individual service. Each classified event + service is a data point in tensor space, and each memory use pattern + service is a data point in tensor space. The vector distance is used to train the correlation between any data point in the tensor space. The idea is that the producer's event may correlate with its own subsequent memory use, but it could also correlate more strongly with another specific service's memory use.

The microservice architecture consists of several services working together and passing several events between them and these events are logged in some event logger or some log files. The examples of such events would be: Service Started up Successfully at 12:02 hours; Hibernate Models Loaded at 13:01 hours; Cache Initialized with data at 14:20 hours.

The solution profiles the microservice's memory and resource utilization at various intervals of time. These snapshots can be logged in "system logs" periodically. The examples of such system logs would be as follows: (a) used heap, (b) allocated heap, (c) CPU usage, (d) garbage collection pause times, (e) garbage collection algorithm used, and/or (f) garbage collection stats. These values are also captured in the system logs at various times (*e.g.*, at 12.02 hours).

The idea is a novel approach to correlate the events logs mentioned above with the system logs on a temporal space and train the application to understand the system resources and used for various events. The trained system can help make a prediction of: when should the virtual machine perform a full garbage collection (*e.g.*, this will help to give the over-allocated heap back to the operating system (OS)), dynamic garbage collector algorithm to be applied, and/or dynamically predict memory and performance issues.

The present techniques may be explained with reference to the examples below:

**Example 1: Prediction of Java virtual machine (JVM) to perform a full garbage collection to reduce over-allocated memory**

Problem: We observed that G1 garbage collection allocates high memory when there is an abrupt change in load. One of the use cases where there is an abrupt increase in load is during service startup. We have observed most of the services had a high allocated memory just after service startup. The issue is this allocated heap is not released back to the OS even though the used heap is low after startup. This was identified as a part of the data training.

Solution using Machine Learning: The solution uses the novel approach of using the training data from the event logs and system logs and correlates the data between the event and system data on a temporal basis to perform full garbage collection. This helps release the over-allocated memory back to the OS. Consider that microservice A has started successfully and micro service B has started successfully. We now have the training data after such correlation on a temporal basis shown in Table 1 below.

| Event Message | Time | Used Heap (MB) | Alloc. Heap (MB) | Full GC | Max GC Pause | Total Heap (MB) | CPU Usage % (Average) | Alloc. Heap Ratio |
|---|---|---|---|---|---|---|---|---|
| Service 1 restarted | 12:02 | 100 | 300 | 0 | 0 | 2000 | 10 | 33 |
| Service 2 restarted | 13:05 | 150 | 300 | 0 | 0 | 2000 | 12 | 50 |
| Service 1: Hibernate Models load | 12:04 | 900 | 1600 | 1 | 5 | 2000 | 14 | 56 |
| Service 2: Hibernate Models load | 13:08 | 850 | 1600 | 1 | 6 | 2000 | 18 | 53 |

*Table 1*

This training data is a very small extract of the learning data. The system learns that after service startup the ratio between used heap and the allocated heap is low and there is a memory which can be returned back to the OS. The system now can perform a full garbage collection to return the extra allocated memory but it needs more data to understand what would be the optimal time to perform the full garbage collection. It now understands for which events the "CPU usage" is low and what events would contribute to a low CPU. Once it identifies an appropriate time when the CPU usage is low, the other parameter to understand is latency. When a full garbage collection is performed, the application latency increases as the application threads are blocked. We use the "max pause GC" parameter to make this decision. This idea is that the max pause GC time shows how much time is required for a full garbage collection and the system should be idle and have low CPU load for a time equal to "max pause GC". When the trained system finds such an idle time then the system will make a decision and perform a full garbage collection and release the memory back to the OS.

Advantage: The over allocated memory was released back to the OS and the application latency was not hampered.

**Example 2: Dynamic Garbage Collector Prediction**

Problem: The prior art solutions show that garbage collection is highly dependent on the behavior of the application as well as on the available resources. It also shows that some garbage collector algorithm is optimal for some heap sizes and at different CPU loads of the system. There is a use case for smaller heap where parallel garbage collection might be the best fit and for higher heap sizes where G1 garbage collection might be a better fit. There are some prior art techniques which use machine learning to predict such algorithm behavior.

Solution using Machine Learning: The techniques presented herein provide a solution where the machine learning algorithm uses the event logs generated by microservices and the system logs generated at snapshot intervals and then correlates the event and resource data to predict the dynamic garbage collector algorithm. In contrast, the prior art solution uses only system logs to make such a decision and there is no correlation or learning based on the events performed by microservices. The correlation between the events and the system logs helps to predict a more accurate outcome of which garbage collector algorithm is the right fit for the particular scenario like varying heap sizes and CPU load.

Consider the training data shown in Table 2 below.

| Event Message | Total Heap Size (MB) | Used Heap | Alloc. Heap | CPU Usage % | Full GC Count | Mac Pause GC (ms) | Throughput | Algorithm |
|---|---|---|---|---|---|---|---|---|
| Service 1 sends a message using Object A | 1000 | 250 | 800 | 26 | 4 | 5 | 99 | G1GC |
| Service 2 sends a message using Object A | 4000 | 1900 | 3400 | 10 | 6 | 6 | 99 | G1GC |
| Service 1 sends a message using Object A | 1000 | 250 | 350 | 18 | 7 | 3 | 98.5 | Parallel GC |

*Table 2*

The solution according to the techniques described herein uses the above example data to train and understand which algorithm will be optimal for various services and under various load. The application at a particular point in time will be optimized either for latency, memory or throughput. We can choose dynamically what is currently important for the application out of the three parameters mentioned above. The trained system will predict accurately which algorithm will be best suited for latency or memory or throughput.

Advantage: The system can dynamically change the garbage collector algorithm and be best optimized for any parameter like memory, latency or throughput at any point in time.

**Example 3: Dynamically predict memory and performance issues**

Problem: The current state of the art solutions cannot accurately predict when an out of memory is likely or what events are likely to take up a lot of memory. The state of the art solutions also cannot accurately predict scheduled executions and how much memory that would consume at scheduled points in time.

Solution using Machine Learning: The solution according to the techniques described herein may be described in reference to Figure 2 below.

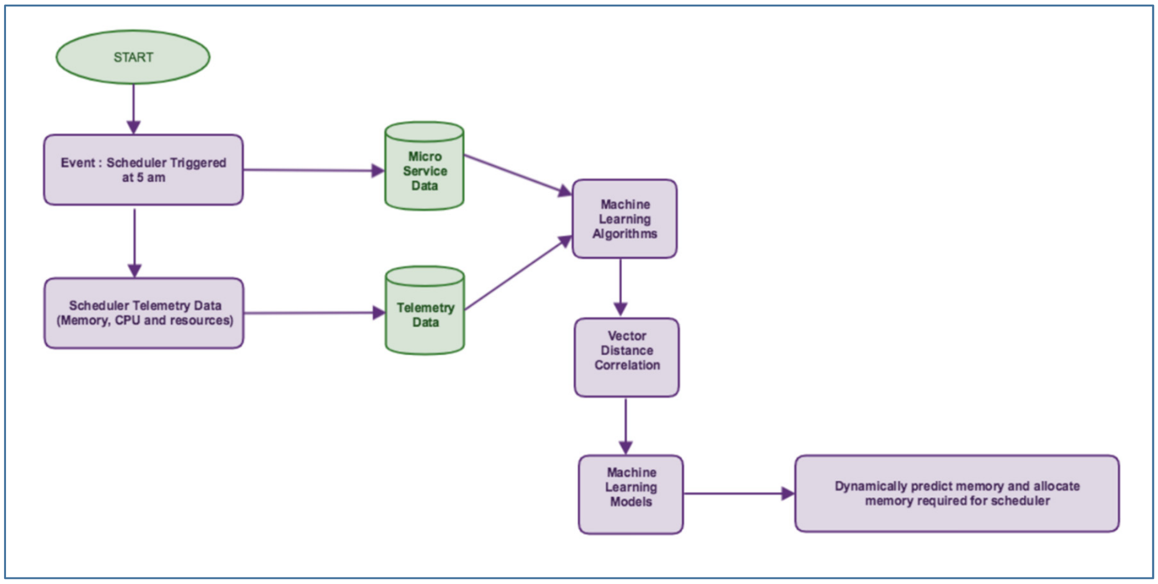
START
Event : Scheduler Triggered at 5 am
Scheduler Telemetry Data (Memory, CPU and resources)
Micro Service Data
Telemetry Data
Machine Learning Algorithms
Vector Distance Correlation
Machine Learning Models
Dynamically predict memory and allocate memory required for scheduler

*Figure 2*

The solution described according to the presented techniques correlates event logs from microservices and system logs of memory and resources and learns that certain events when triggered lead to a lot of memory increase. Consider that there is a memory leak in the system and it happens when microservice 1 sends a message to microservice 2. The system learns after training that there is a memory which is increased when a Message is sent from microservice 1 to 2. It can predict such behavior and alert to fix such issues.

The system can co-relate the scheduled event from “event logs” and correlates the resources required to that scheduled event. For example, say microservice 1 has a scheduled execution at 5 am every day to backup all of its data to another data store. This scheduled execution takes up a lot of memory. The system learns using the trained data that the execution occurs at 5 am every day when the “scheduled event” is received. The system adapts and is ready with the memory and resources as the event is received and once the scheduled event is completed, it will perform a full garbage collection and release the allocated memory using the solution described above for Example 1 (*i.e.*, Prediction of JVM to perform a full garbage collection to reduce over-allocated memory).

Advantage: The system can better accurately predict the memory changes and be prepared for the events where there is high load and relinquish the memory back on lower loads.