Investigating the Effect of Garbage Collection on QoS of Multitenant PaaS Clouds

MSc Research Project
Cloud Computing

Nikhil Kumar Satish Kumar Sharma
x17110513

School of Computing
National College of Ireland

Supervisor: Vikas Sahni

Student Name:	Nikhil Kumar Satish Kumar Sharma
Student ID:	x17110513
Programme:	Cloud Computing
Year:	2017
Module:	MSc Research Project
Lecturer:	Vikas Sahni
Submission Due Date:	13/08/2018
Project Title:	Investigating the Effect of Garbage Collection on QoS of Multitenant PaaS Clouds
Word Count:	4473

Signature:	
Date:	16th September 2018

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Investigating the Effect of Garbage Collection on QoS of Multitenant PaaS Clouds

Nikhil Kumar Satish Kumar Sharma
x17110513
MSc Research Project in Cloud Computing

16th September 2018

Abstract

A PaaS Cloud is expected to provide an environment for users to install and deploy the application in which each tenant is isolated from co-tenants and one tenant is not allowed to interfere with the performance of the co-tenants by over-utilizing of hardware resources. PaaS tenants most often run on shared Virtual machines or physical nodes whose performance is dependent on the underlying resource limitations as well as the actions performed by other tenants. The cloud service provider is expected to maintain certain Quality of Service (QoS) as per the SLAs. PaaS cloud services are implemented in high-level languages like Java which provide features for automatic memory management policies such as Garbage Collection (GC) which can affect the QoS. The language runtime implements various policies and algorithm to conduct a GC which can hamper the servers ability to maintain accepted levels of QoS. This paper attempts to investigate which GC policies performs best and has lowest interference in a PaaS cloud environment.

1 Introduction

Platform as a service (PaaS) a cloud-based service model that forms a abstraction of the application stack level functions such as runtime, frameworks, build packs and middleware together to form a service. It thus provides a platform for developers or tenants to deploy the application Kavis (2014). For example, PaaS provide solution and services like caching, asynchronous messaging, scaling that a developer can focus on building the business logic. The tenants have control over the deployed applications and is less concern over cost and management of the underlying infrastructure. The major cloud based PaaS providers include IBM Cloud, Heroku by Salesforce , Red Hat Openshift, Amazon Elastic Beanstalk, Microsoft Azure and Google App engine as discussed by Patros et al. (2016).

Multitenancy empowers computing resources to be shared between multiple tenants and is core feature of cloud computing. The tenants execute their application on cloud assuming to hold the resources discretely. According to Patros et al. (2015) resource sharing according to cloud computing principles mentioned by Kavis (2014) each tenant should be isolated from co-tenants in a way they do not over consume each others resources. Tenants presume that the resources are unlimited in cloud allowing them scaling up compute, memory and storage but in a public cloud the resources are shared, and performance is dependent on resources allocated and resource utilization of other tenants.

PaaS cloud services are implemented on platforms such as Cloud Foundry which support high-level languages like Java, GO. C# and ruby, which provide an automatic memory management features and policies, also known as Garbage Collection (GC). The Cloud service provider ideally should maintain a Quality of Service (QoS) for performance as per the Service level objectives, otherwise can lead to nancial penalties as discussed in He et al. (2014)). According to Jones et al. (2016) GC is potentially a high computational task and can interfere with the JVM ability to function optimally thereby affect servers ability maintain its accepted level of QoS. A complete GC can completely pause/stop the threads request handling from time to time free up memory. Thus, measuring the impact GC policies have on various applications, runtime and tenants in multi-tenant PaaS environment is crucial.

According to Patros, Kent and Dawson (2017) in a production applications deployed in PaaS, has conducted research on various resource interference and scaling in PaaSPatros et al. (2016). This paper further attempts to research on stressing the GC component of the runtime in a direct and controllable way. Thus, proposal to implement and extend a congurable PaaS application framework that aims in stressing the GC component of the underlying runtime by capturing and evaluating the GC stats. The investigation will experimentally evaluate the performance of the four GC policies (Gencon, Balanced, Optavgpause and Optthroughput) available in the IBM J9 Java runtime in a multitenant environment specially IBM Cloud based on Cloud Foundry services. It will evaluate based on the test results, which GC policy incurs the lowest impact to co-located tenants and benchmark, identify the best policy to maintain QoS.

2 Literature Review

2.1 Multitenancy on PaaS Cloud

Multitenancy allows resources to be shared between tenants and helps to reduce cost, improve hardware resources and energy as discussed by Rimal et al. (2009). In a multitenant environment one tenant ideally should not be allowed to read or modify data of another tenant. Also, tenants should not be allowed to take over consume resources such a CPU and memory that would hamper the performance of other applications. Platform as a Service (PaaS) is a type of Cloud Computing service model which provides a platform for the applications to be developed and run by eliminating the dependencies required for creating infrastructure and maintenance as mentioned by Kavis (2014). Cloud service providers offer a SLA for the services, wherein inability to maintain it could lead to financial penalties. Thus, as mentioned by Patros, Dilli, Kent and Dawson (2017) cloud services providers need to maintain the QOS mentioned in the SLA. The investigation on resource interference and scaling in a multitenant PaaS environment has been conducted by Patros et al. (2016) which would be discussed futher.

2.1.1 Benchmarking Cloud Tenants

Benchmarking is critical in evaluating the questions raised in concern with the performance interference, scaling and location of the tenant in a standardized manner. Thus, Patros et al. (2016) implemented cloud burners which replicates the behavior of cloud tenants. It selects each resource such as CPU, cache, resident memory, network I/O, disk I/O which is stress loaded which starves other tenants from those resources impacting performance. The Cloud burner which is a Java EE application was successful in targeting selected hardware resources whereby all the test registering cache misses. The testing methodology was looped for duration of 120 seconds for each resource, the burner is thus capable of targeting the resource it intends to. Thus, it can be used to evaluate the performance interference in between cloud tenants in a structural manner as it can explicitly measure and target consumption of the independent resources as demonstrated by Patros et al. (2016).

2.1.2 Resource Interference and Scaling using Cloud Burners

Patros et al. (2016), by using the Cloud Burner which was discussed above, propose a plan to evaluate the performance interference between cloud tenants for profiling the application based on resourced based slowdown analysis. Additionally, scaling the CPU for tenants on same VM was performed which showed no significance effect on its performance while scaling out in the case of multiple VM it was found that CPU intensive tenants have performance interference on cotenants. The paper did not highlight resource targeting for databases which would be detrimental for application profiling in cloud that can lead to interference's.

2.2 Java Virtual Machine Isolation

Java Virtual Machine (JVM) which is a part of Java Runtime Environment (JRE), provides platform independence making it portable to run on any hardware or machine with a JRE, as described by founders of JVM Lindholm et al. (2014), it is also used in containers build packs for running Java applications on cloud. JVM provides tenant isolation on a single JVM because of Java specifications class loaders feature for partial security isolation. The tenants in JVM are isolated in way that one tenant is locked to read or write object of another tenant unless explicit reference to the objects are specified. The Java classes having similar class name shared between multiple tenants does not cause any interference due to the way in which Java class loaders are designed. There would not be any obscurity as long as every tenant has references to its own class loader as disused in the paper by Patros et al. (2015). Apart from the benefits, the drawback of the JVM isolation would be further discussed.

In Java Enterprise Edition an Enterprise Archive (EAR) is used for mapping multiple components into a single component providing a feature for Java class library files (JCL) to be loaded by system class loader that implies there is no separation even with different class loaders. Patros et al. (2015) discussed in order to increasing sharing, the tenants could use a common JCL, but this could lead to another issue allowing modification of static fields which is violation of multitenancy isolation required. The above implications can cause deadlocks and lock contentions on common class objects. Thus, apart from providing security isolation, performance isolation and resource isolation is not provided by JVM. A tenant inside the sharing the same JVM could over utilize resources explicitly as discussed by Patros et al. (2016) and also crashing of one tenant could recursively impact all the tenants sharing the JVM due to interference.

2.2.1 Mulititenant Runtime as a Service:

Runtime as a Service provided by IBMs JVM solves the issues related to tenant sharing and isolation. In RaaS configuration is made to store the static fields and respective monitors related to each tenant. In a Application server the memory implication can be divided into three parts namely the memory required for Java run time i.e. the heap memory, memory required to run the underline OS and VM, and lastly memory required by tenant applications. To save and optimize memory utilization for usage of application server a new theoretical model is proposed by Patros et al. (2015) namely Application Server as a Service (ASaaS). By sharing application data between tenants using existing RaaS, a tenant isolation can be achieved using ASaaS. The memory footprint is drastically reduced by 36% compared to 60% in RaaS model when multiple tenants are considered but the drawback is that there is 10% decrease in memory footprint for data intensive tasks. While signification saving in memory is proposed by this model but the implications of Automatic memory management such as GC was not considered which could impact throughput and response time of the services.

2.3 Garbage Collection

2.3.1 JVM tunning and Heap allocation

The heap allocated to the JVM instance is shared by all the application threads installed on top of it. The heap allocation is initiated right after JVM instance is started that is used by objects allocation for automatic memory management. The automatic memory management policies are not presumed but chosen depends on the behavior of the application. The default JVM configuration is good for application lower users count and memory footprint but the JVM needs to tweaked for as the needs increases. The JVM allows runtime configuration to be configured in an application server wherein initial and max heap can be specified, whenever those values are modified the JVM instance needs to be restarted for allocation of new heap size. This type of heap allocation posses a threat when application needs exceed the heap allocation during runtime for execution, slowing down resources and a OutOfMemory error is thrown as mentioned by Jones et al. (2016).

2.3.2 Garbage Collection and Performance tuning

The JVM heap size consists of two parts: growth size of heap and time required for Garbage Collection (GC). Garbage collection in a JVM clears the unused objects and allocations space for new objects to be stored and retrieved by the application. Thus GC in Java destroys the objects which are not in use by running continuously in the background, thus it can be said that goal of garbage collector is to destroy the objects which are not reachable as discussed by Bruno and Ferreira (2018). According to Jones et al. (2016) the GC polices need to be optimized for various scenarios, for example a GC overhead is greater that 10 % then policy has to be changed. A GC occupancy of more that 75 % can lead to high GC cycles and a occupancy less than 40 % means longer garbage collection cycles which generally may impact the performance. An out of memory exception means the heap size is too small to hold the objects and the heap size has to be increased such as Minimum heap size (-Xms) and Maximum heap size (-Xmx).

GC configuration can help to improve the health and performance of application running in a server, the JVM arguments and policies thus play a detrimental part for profiling Java application. Jones et al. (2016) in his book says that GC is often resource intensive as it needs to move objects graphs of the threads and evaluate the objects to be deleted from the heap. Patros, Kent and Dawson (2017) describes GC has ability to hamper the QoS defined by the SLAs. This could thus , also impact other cotenants in multitenant environment as complete GC could stop or slow down the all the threads execution to clean objects and heap space.

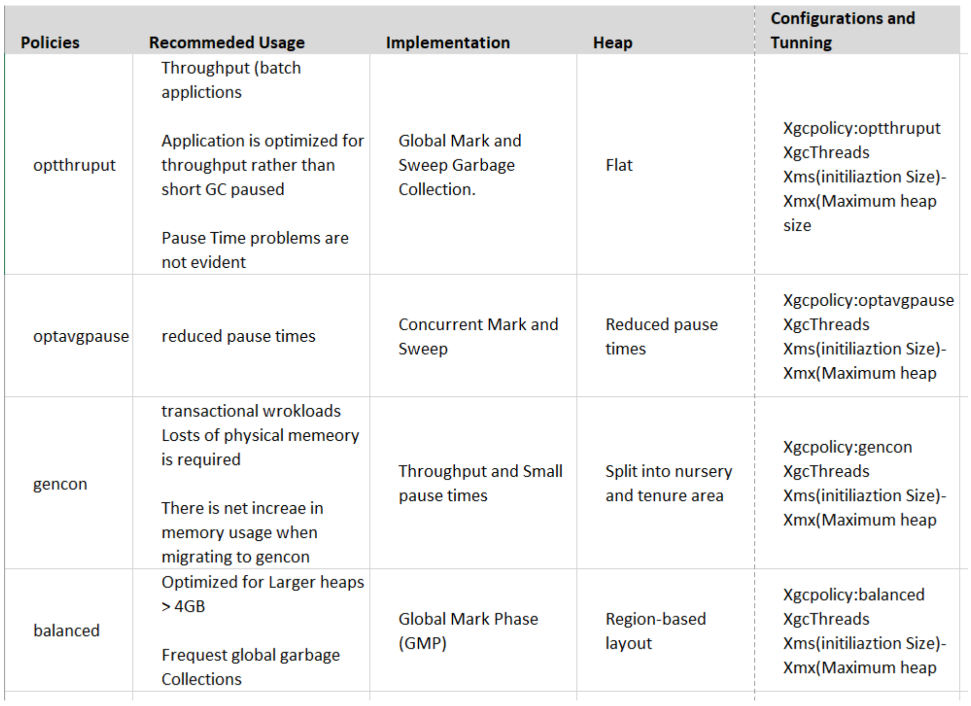
2.4 CloudGC : A need for GC benchmarking

Patros, Kent and Dawson (2017) highlights need to investigated the problem of Service level objectives satisfaction on cloud services due to GC. Previous benchmarks on

Service oriented Java EE that include Cloud store by Lehrig et al. (2018), other Java benchmarks such as by Baylor et al. (2000) and Cloud burners which was previously discussed formulated by Patros et al. (2016) which focuses on specific hardware resources do not explicitly focus or target on GC component of the runtime as highlighted by Patros, Kent and Dawson (2017). Thus, a proposal is made to implement CloudGC which PaaS application framework with ability to be configurable that stresses on GC component of the runtime.

Patros, Kent and Dawson (2017) implements this CloudGC to evaluate four GC policies found in IBM J9 runtime namely by Bailey et al. (2011) and Sciampacone et al. (2011):

- Gencon
- Balanced
- Optavgpause
- Optthroughput.



Policies	Recommeded Usage	Implementation	Heap	Configurations and Tunning
optthruput	Throughput (batch applictions Application is optimized for throughput rather than short GC paused Pause Time problems are not evident	Global Mark and Sweep Garbage Collection.	Flat	Xgcpolicy:optthruput XgcThreads Xms(initiliaztion Size)- Xmx(Maximum heap size
optavgpause	reduced pause times	Concurrent Mark and Sweep	Reduced pause times	Xgcpolicy:optavgpause XgcThreads Xms(initiliaztion Size)- Xmx(Maximum heap
gencon	transactional wrokloads Losts of physical memeory is required There is net increae in memory usage when migrating to gencon	Throughput and Small pause times	Split into nursery and tenure area	Xgcpolicy:gencon XgcThreads Xms(initiliaztion Size)- Xmx(Maximum heap
balanced	Optimized for Larger heaps > 4GB Frequest global garbage Collections	Global Mark Phase (GMP)	Region-based layout	Xgcpolicy:balanced XgcThreads Xms(initiliaztion Size)- Xmx(Maximum heap

Figure 1: GC Policies Comparison Bailey et al. (2011) and Sciampacone et al. (2011)

Patros, Kent and Dawson (2017) implemented a GC focused benchmarking tool which was deployed on PaaS cloud platform cloud foundry that investigate the four GC polices in local and isolated environment. The configuration settings CLoud GC offered parameters to be configured during the runtime. Thus, implementation provides room for broad number of configurations. The captured inputs such throughput and response time metrics, were processed afterwards to get the results. Apart from the GC stressing the paper does not consider performance impact through resource interference and scaling, and other tenants as it could largely effect the results of investigation.

3 Methodology

The PaaS cloud services provide run-times and build packs for running cloud based application, one such offering used by most cloud providers is based on Cloud Foundry Foundations. As discussed by Patros et al. (2016), various hardware resources that are shared by multiple tenants are shared by multiple tenants and interference can slow down other tenants sharing those resources. Apart from various hardware resource Patros, Kent and Dawson (2017) highlights lack of GC bench-marking tool for PaaS cloud which exclusively targets all the GC resource as it can halt services when a GC occurs. Thus, identifying GC policies, Optimized parameters such as heap size is critical in cloud environment.

In this Research Project, the work done by Patros, Kent and Dawson (2017) is further extended to benchmark GC components on public cloud offerings like IBM cloud as the previous research was performed on an isolated and local instance of Cloud Foundry with liberty build pack. This project captures the used heap, average response time in seconds, throughput and GC stats. It captures and analzyes heap size, pause times, heap before and after collection, sweep time. The GC stats are further extended to be captured by Python script and stored locally whenever the stats are updated. This will help to capture and analyze additional GC components such GC policy, GC Collection Count, GC Collection time.

3.1 IBM Cloud and Cloud foundry

IBM Cloud provides a catalogue of services on PaaS platform build on its value-added distribution of Cloud Foundry services to provide build packs of all major runtimes including liberty build pack of Java as discussed in Bernstein (2014). It provides compute resources along with application servers as a PaaS service. This allows applications to be quickly deployed on a public cloud with application management features included. Considering the Java EE runtime offering IBM cloud becomes ideal for benchmarking Cloud GC application as it is built on the IBM JVM runtime J9. IBM also provide a single tenant based Bare Metal server for HPC and I/O intensive tasks and workloads as disscsueed in Kim et al. (2016).

3.1.1 IBM WebSphere Application Server on Cloud

IBM WAS liberty core plan, IBM WebSphere Base plan, IBM WebSphere ND plan to provide customizable and full feature Application server. A single instance of WAS Base plan with 4096 MB main memory and 2 virtual CPUs is allocated. The Application server consists of a configurable profile hosting the JVM to run installed applications on top of it. For the Cloud GC application to run heap values are defined in the general properties of the JVM with Initial Heap size set to 2048 mb and Maximum heap size set to 80% of the host machine i.e. 3276 mb. The generic JVM arguments are used to specify the policy for GC, by default a global policy gencon is used by the application server. The Application server comes with a Integrated Solutions control where the configurations is made and war file of application is deployed. The application server consists of only single application Cloud GC running and installed to avoid any interference from other applications.

OpenVPN is used to connect to the Integrated console of WebSphere and Host VM through Putty using SSH connection, A VPN connection is established between local machine and IBM cloud for security and servers cannot be accessed from outside with it. IBM cloud provides a feature for Public IP to be assigned to the host machine, after requesting the public IP the access status is kept open for it to be mapped with host-machine. This public IP can also be mapped to a DNS server. The Application is accessible using a Public IP after configuring the Endpoint of HTTP inbound requests in WebSphere Application server.

3.2 Load Test Plan using Apache JMeter

Apache JMeter is used to load test the functional behavior of the Java EE application, here it used to create a load of Number of Threads (users): 1,8 and 64 with a Ramp-Period (in seconds) of 2,16 and 128 seconds which means after every 2 seconds 1 thread is started. The Loop count is set forever, and the scheduler is enabled with scheduler configuration to set Duration (seconds) as 300s, a start up delay is set to 10 seconds before the test plan executes. A HTTP Request is sent to the webserver with the host IP address of the IBM cloud machine running on Red Hat linux and application server with default host 9080 hosting the application services. The HTTP request is send using POST method here on the path /GrapAction to stress test which generates the HTTP Servlet Request and Response recursively. A HTTP default request is also sent to the root directory of the application over the same port and hostname for each instance in this case JVM instance server on IBM WebSphere Application Server Base Edition.

3.3 GC Benchmarking Framework

Cloud GC is Java EE application created to be run on PaaS cloud environment specially for IBM bluemix Runtime. It can configure various GC parameters to reproduce GC patterns on the runtime. This can useful for performing load tests on runtime, various GC policies like Gencon, Balanced, Optavgpause, Optthroughput of IBM JVM, Application performance, this can be useful it terms of research and benchmarking enterprise application on cloud. Apache JMeter is used to stress test the environment, it can be configured as to replicate the real world scenario creating standard test cases. It can highlight the GC component of runtime which would determine the performance of the policies having the lowest impact in public cloud thereby having lowest impact on other tenants.

Cloud GC consists of and Initialization component which initializes the GC parameters, the initialized values can be used to start and benchmark the GC component. The object graphs used by GC are displayed using a graph. The servlets are exposed for request initialization of graph using a HTTP POST request from outside the network.

The metrics captured and plotted on the graphs include :

- Used Heap(mb): Plots the utilized heap memory
- Average Response Time: Plots the response time of each thread.
- Average Throughput: Plots the throughout of each requests.
- Reclaimed per GC: Shows the graph of reclaimed GC.
- Requests per GC: Plots numbers of GC requests.

To recreate functional components of a real world or production applications, stack structure is maintained by Cloud GC serving as the object graphs roots that is initially developed by Patros, Kent and Dawson (2017). When a request is made new object are incremented over the stack and object reference to each other is made for the previous stack. It keeps the object locked to the request scope, the objects are are aloud to die young, as per the ideas of GC in Jones et al. (2016), this allows a stateless software design architecture.

The GC parameters provided by the Cloud GC application including addition and removal of frame, traverse. The parameters can be dynamically executed during run-time using load-test at the servlet endpoint. This allows wide selection of parameters to be configured during the run-time for various test case scenarios related to GC components and bench-marking them the results.

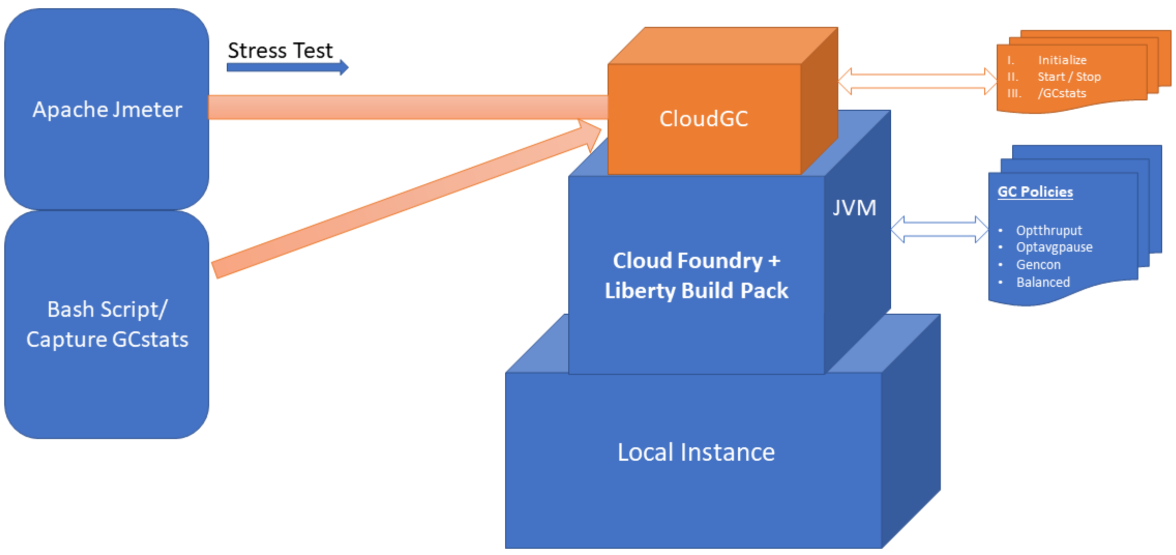


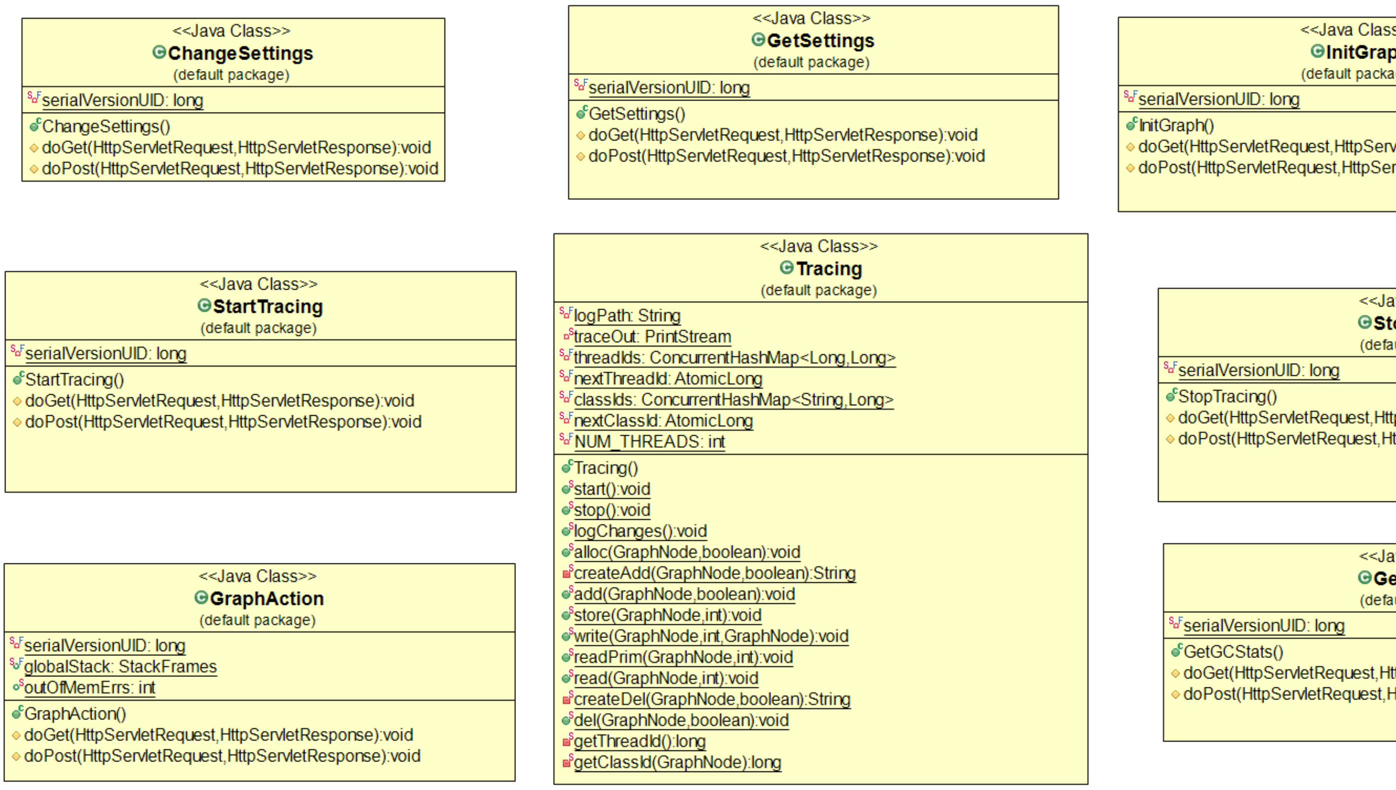
Figure 2: Cloud GC Architecture

4 Implementation

To investigate the performance of the four GC policies in the IBM JVM, the following implementation method was conducted : The application was deployed on deployed IBM cloud with cloud foundry services. IBM WebSphere Server Base edition with single server instance was installed on top of it. The polices were configured to it for each tests, verboseGC was enabled, initial and maximum heap size of 2048 mb and 3276mb respectively was initialized. To stress the Cloud application a stress tests of 1,8 and 64 threads/users were fired on loops using Apache JMeter for a duration of 300 seconds each, the timeout was set to 2 seconds for each thread. The load was driven from a local machine outside the IBM cloud environment.

In all the cases the Initializing parameters were Set to default values , the heap was set constant and the load was set to constant. For each test the JVM was restarted. The server was also constant with 4096 main memory and two virtual cpu cores. The GC stats values of all the JVMs were captured using recursive python script and save local machine for every GC count. All the JMeter stress test HTTP requests were successfully hit to the application. The GC stats were further analyzed for detailed analysis to determine the optimal GC policy having the lowest impact on cloud.

4.1 Cloud GC classes



<<Java Class>>
ChangeSettings
(default package)
serialVersionUID: long
ChangeSettings()
doGet(HttpServletRequest,HttpServletResponse):void
doPost(HttpServletRequest,HttpServletResponse):void
<<Java Class>>
GetSettings
(default package)
serialVersionUID: long
GetSettings()
doGet(HttpServletRequest,HttpServletResponse):void
doPost(HttpServletRequest,HttpServletResponse):void
<<Java Class
InitGrap
(default packa
serialVersionUID: long
InitGraph()
doGet(HttpServletRequest,HttpServ
doPost(HttpServletRequest,HttpSer
<<Java Class>>
StartTracing
(default package)
serialVersionUID: long
StartTracing()
doGet(HttpServletRequest,HttpServletResponse):void
doPost(HttpServletRequest,HttpServletResponse):void
<<Java Class>>
Tracing
(default package)
logPath: String
traceOut: PrintStream
threadIds: ConcurrentHashMap<Long,Long>
nextThreadId: AtomicLong
classIds: ConcurrentHashMap<String,Long>
nextClassId: AtomicLong
NUM_THREADS: int
Tracing()
start():void
stop():void
logChanges():void
alloc(GraphNode,boolean):void
createAdd(GraphNode,boolean):String
add(GraphNode,boolean):void
store(GraphNode,int):void
write(GraphNode,int,GraphNode):void
readPrim(GraphNode,int):void
read(GraphNode,int):void
createDel(GraphNode,boolean):String
del(GraphNode,boolean):void
getThreadId():long
getClassId(GraphNode):long
<<Ja
Sto
(defa
serialVersionUID: long
StopTracing()
doGet(HttpServletRequest,Htt
doPost(HttpServletRequest,Ht
<<Java Class>>
GraphAction
(default package)
serialVersionUID: long
globalStack: StackFrames
outOfMemErrs: int
GraphAction()
doGet(HttpServletRequest,HttpServletResponse):void
doPost(HttpServletRequest,HttpServletResponse):void
<<Ja
Ge
(defa
serialVersionUID: long
GetGCStats()
doGet(HttpServletRequest,Ht
doPost(HttpServletRequest,H

Figure 3: Class Diagram of Cloud GC

- Change Settings: This class changes the parameters associated with GC for initialization.

- Init Graph: Initializes the graph appends the run time with free memory.
- Tracing: Plots the concurrent hash map on the graph.
- Start Tracing : Starts the initialized parameters and displays the graphs after tracing
- Stop Tracing: Calls the servlet to stop tracing
- Graph Action: Servlet used to receive stress test
- GC stats: Servlet to process and capture GC related stats.

4.2 Implementation and Stress Testing Process

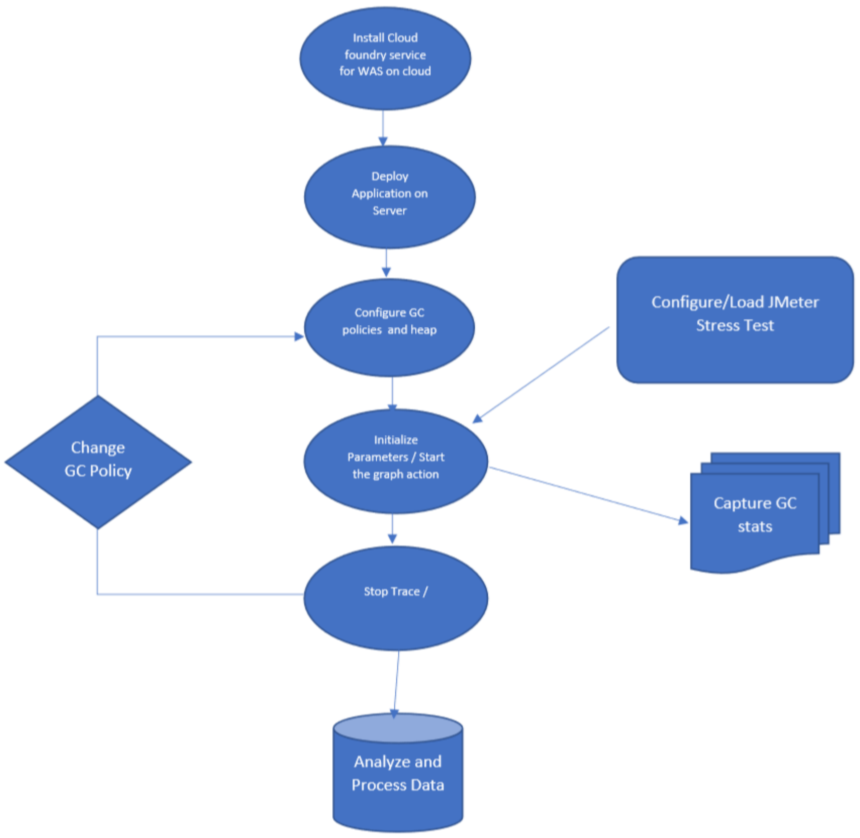

Install Cloud foundry service for WAS on cloud
Deploy Application on Server
Configure GC policies and heap
Configure/Load JMeter Stress Test
Change GC Policy
Initialize Parameters / Start the graph action
Capture GC stats
Stop Trace /
Analyze and Process Data

Figure 4: Flow Diagram of the Implement solution

1. Install new set of cloud foundry service with setup of IBM Websphere Application server.

   Deploy the application and change the GC policy to -Xgcpolicy:gencon but adding the JVM parameters on DMGR console and initialize the heap. (change to each policy)

2. Fire setup request to Servlet Change Settings to initialize Cloud GC parameters.
3. Start the tracing process and fire load with JMeter to stress test.
4. Run the python script to capture the GC stats with a request at /GetGCStats.
5. Further process the data for analysis and evaluation.

5 Evaluation

As Patros, Kent and Dawson (2017) used cumulative frequency to plot a graph and evaluate the performance of all the GC polices, this research project further extends it to capture GC stats such as heap size, Free heap before and after collection, Amount of GC freed, Pause times and Sweep time to further analyze and compute optimal GC policy for PaaS environment. Mean, Minimum and Maximum values of the above mentioned parameters are captured and plotted against each other for all the four JVM policies.

5.1 Heap Size

Table 1: Heap Size

Variant	Mean	Minimum	Maximum
	heap (GB)	heap (GB)	heap (GB)
balanced	3.09	2	3.2
gencon	2.67	2	2.97
optavgpause	3.09	2	3.2
optthruput	3.14	2	3.2

From the table heap size and graph Figure 5 it can be observed that gencon has the lowest mean heap and is the most performing, while balance and optavgpause have same scores and opthruput has the most heap utilized.

5.2 Free heap (after collection)

Table 2: Free heap (after collection)

Variant	Mean	Minimum	Maximum
	heap (GB)	heap (GB)	heap (GB)
balanced	1.33	0.56	1.98
gencon	0.9	0.49	1.72
optavgpause	1.44	0.78	1.97
optthruput	1.29	0.21	2.15

From Figure 6 and Figure 7 it can be observed that free heap before and after collection gencon has the lowest mean heap and hence is the most performing followed by balanced, optavgpause and optthruput respectively.

5.3 Free heap (before collection)

From Table Before Collection and Figure 7 it can be observed that gencon occupies the least amount of heap whereas balanced has the highest mean heap throughout.

Table 3: Free heap (before collection)

Variant	Mean	Minimum	Maximum
	heap (GB)	heap (GB)	heap (GB)
balanced	0.81	0.16	1.92
gencon	0.37	0.1	1.67
optavgpause	0.13	0	1.92
optthruput	0.15	0	1.92

5.4 Amount Freed

From the Amount freed table and Figure 8 it can observe that, gencon is capable of freeing maximum memory, followed by balanced optthruput and optavgpause.

Table 4: Amount Freed

Variant	Mean	Minimum	Maximum	Total
	heap (GB)	heap (GB)	heap (GB)	heap (GB)
balanced	0.52	0	0.96	175
gencon	0.53	0.05	0.9	223
optavgpause	1.3	0.04	1.64	91.1
optthruput	1.12	0.02	1.99	75.2

5.5 Pause times

Table 5: Pause Times

Variant	Mean	Minimum	Maximum	Total
	time (seconds)	time (seconds)	time (seconds)	time (seconds)
balanced	0.14	0	6.68	48.9
gencon	0.09	0	1.37	38.3
optavgpause	2.06	0.07	10.8	144
optthruput	2.36	0.06	21.8	158

In terms of pause times similar performance is observed from above table and Figure 9 gencon has the lowest amount of pause times overall and the maximum time required for gencon to perform a pause for GC is lowest as well, this translates to an optimum throughput. optavgpause and optthruput have higher pause time which could have impact on other tenants in cloud.

5.6 Sweep Time

In terms of Sweep time from Figure 10 gencon has the lowest sweep time but optavgpause matches it up in terms of mean and maximum time.

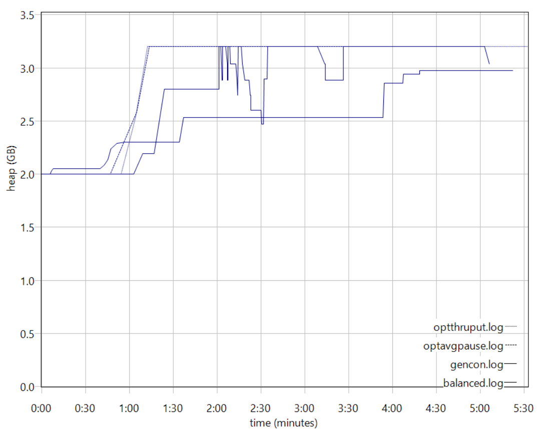


Figure 5: Heap Size

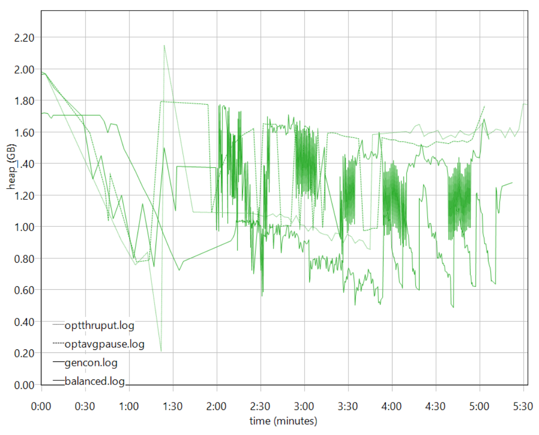


Figure 6: Free Heap (after collection)

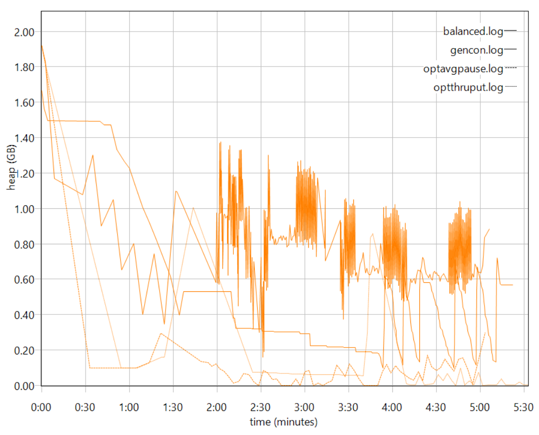


Figure 7: Free Heap (before collection)

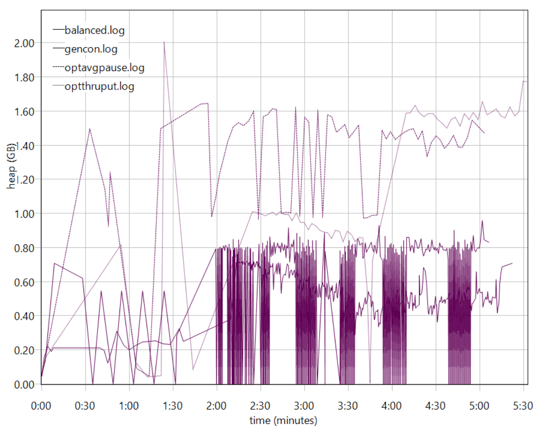


Figure 8: Amound Freed

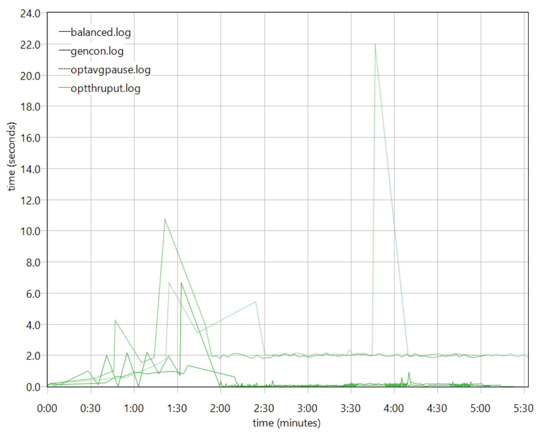


Figure 9: Pause Times

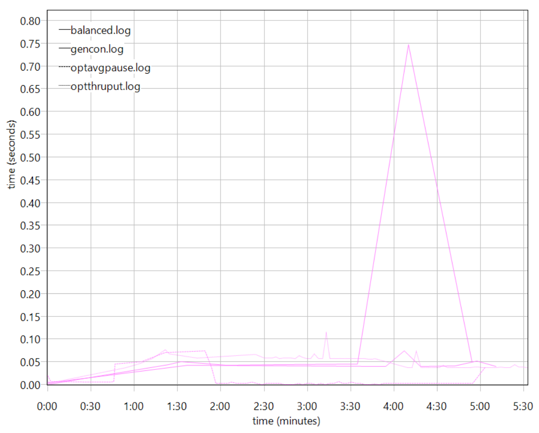


Figure 10: Sweep Time

Table 6: Sweep Time

Variant	Mean	Minimum	Maximum	Total
	time (seconds)	time (seconds)	time (seconds)	time (seconds)
balanced	0.11	0	0.75	1.02
gencon	0.04	0	0.07	0.34
optavgpause	0.01	0	0.07	0.57
optthruput	0.05	0	0.11	3.3

5.7 Discussion

Table 7: Summary

GC Mode	balanced	gencon	optavgpause	optthruput
Proportion of time GC pauses (%)	16.28	12.05	47.31	47.31
Proportion of time unpaused (%)	83.72	87.95	52.69	52.69
Rate of GC (MB/minutes)	35043	42419	18466	13909

The GC stats QoS results, which are aggregated over multiple polices and test reveals that Gencon which the default policy of WebsSphere Application server is the most performing in all the test cases that were considered, it has the capacity to perform GC at the highest rate, sometimes 2-3 times faster than other polices like opthruput, also the proportion of GC pauses is lowest. Hence, gencon should be the preferred policy for PaaS cloud services as lower GC pauses would have lowest impact on other tenants. Secondly, balanced was quite comparable to gencon in terms of of pause times and unpaused %. Finally, the last two GC polices optavgpause and optthruput underperfomed compared first two polices.

6 Conclusion and Future Work

In the Research Project, the goal was to identify GC policy in IBM J9 runtime which would cause lowest interference due to GC on Cloud and other tenants, as complete GC can halt all the processes and affect other tenants sharing the resources. A PaaS based application Cloud GC installed IBM cloud with Cloud foundry services : WebSphere Application server base edition. Stress test was performed on the application using Apache JMeter. After running the test and bench-marking the GC parameters, GC stats for all policies were captured. Further, after evaluating the results it was found that Gencon had the lowest pause time, highest GC rate and lowest heap utilization. The GC data and other resource metrics were not captured from other tenants due to limitation of services.

In Future work, the impact on other co-tenants running on PaaS application that are sharing resources can be investigated, due to limited resources available this could not be conducted as it would be quite expensive to simulate this environment on cloud. Additionally, the GC stats captured could be directly connected and send to database such a MongoDB on same or different instance per instance.

References

Bailey, C., Gracie, C. and Taylor, K. (2011). Garbage collection in websphere application server v8, part 1: Generational as the new default policy, *IBM Developer Works* .

Baylor, S. J., Devarakonda, M., Fink, S., Gluzberg, E., Kalantar, M., Muttineni, P., Barsness, E., Arora, R., Dimpsey, R. and Munroe, S. J. (2000). Java server benchmarks, *IBM Systems Journal* **39**(1): 57–81. Core Ranking : A.

Bernstein, D. (2014). Cloud foundry aims to become the openstack of paas, *IEEE Cloud Computing* **1**(2): 57–60.

Bruno, R. and Ferreira, P. (2018). A study on garbage collection algorithms for big data environments, *ACM Comput. Surv.* **51**(1): 20:1–20:35. Core Ranking : A*.

He, H., Ma, Z., Chen, H., Wu, D., Liu, H. and Shao, W. (2014). An sla-driven cache optimization approach for multi-tenant application on paas, *2014 IEEE 38th Annual Computer Software and Applications Conference*, pp. 139–148. Core Ranking : A.

Jones, R., Hosking, A. and Moss, E. (2016). *The garbage collection handbook: the art of automatic memory management*, CRC Press.

Kavis, M. J. (2014). *Architecting the cloud: design decisions for cloud computing service models (SaaS, PaaS, and IaaS)*, John Wiley & Sons.

Kim, M., Mohindra, A., Muthusamy, V., Ranchal, R., Salapura, V., Slominski, A. and Khalaf, R. (2016). Building scalable, secure, multi-tenant cloud services on ibm bluemix, *IBM Journal of Research and Development* **60**(2-3): 8:1–8:12.

Lehrig, S., Sanders, R., Brataas, G., Cecowski, M., Ivanek, S. and Polutnik, J. (2018). Cloudstore towards scalability, elasticity, and efficiency benchmarking and analysis in cloud computing, *Future Generation Computer Systems* **78**: 115 – 126. Core Ranking : A.

Lindholm, T., Yellin, F., Bracha, G. and Buckley, A. (2014). *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st edn, Addison-Wesley Professional. Cited by 5365.

Patros, P., Dilli, D., Kent, K. B. and Dawson, M. (2017). Dynamically compiled artifact sharing for clouds, *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 290–300. Core Ranking : A.

Patros, P., Dilli, D., Kent, K. B., Dawson, M. and Watson, T. (2015). Multitenancy benefits in application servers, *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, IBM Corp., pp. 111–118.

Patros, P., Kent, K. B. and Dawson, M. (2017). Investigating the effect of garbage collection on service level objectives of clouds, *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, IEEE, pp. 633–634. Core Ranking : A.

Patros, P., MacKay, S. A., Kent, K. B. and Dawson, M. (2016). Investigating resource interference and scaling on multitenant paas clouds, *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, IBM Corp., pp. 166–177.

Rimal, B. P., Choi, E. and Lumb, I. (2009). A taxonomy and survey of cloud computing systems, *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, Ieee, pp. 44–51.

Sciampacone, R., Burka, P. and Micic, A. (2011). Garbage collection in websphere application server v8, part 2: Balanced garbage collection as a new option, *IBM WebSphere Developer Technical Journal* .