



Computation Support in DROP

v5.02 19 August 2020



Java 9 Garbage Collection Algorithms

Overview

1. Role of Java Garbage Collection: *Garbage Collection* (GC) is one of the features behind Java's popularity. Garbage collection is the mechanism used in Java to free unused memory. Essentially, it is tracking down all the objects that are still used and marking the rest as garbage.
2. Automated Memory Management in Java: Java's garbage collection is considered to be an automatic memory management scheme because developers do not have to designate objects as being ready to be deallocated. Garbage collection runs on low-priority threads.
3. Concepts and Algorithms behind GC: This chapter goes through the various concepts behind memory allocation/deallocation, the algorithms that run behind the scene, and the options to customize the behavior.

Object Life Cycle

1. Stages in an Object Life Cycle: A java object's cycle can be seen as composed of being in 3 stages.
2. Object Creation: To create an object, the *new* keyword is generally used, i.e.,

Object obj = new Object();

When an object is created, a specific amount of memory is allocated for storing the object. The amount of memory allocated can differ based on architecture and JVM.



3. Object in use: After creation, the object is used by the application's other objects, i.e., other live objects have references pointing to it. During its usage, objects reside in memory, and may contain references to other objects.
4. Object Destruction: The garbage collection system monitors objects and, if feasible, counts the number of references to each object. When there are no more references to an object, there is no way to get to it with the currently running code, so it makes perfect sense to deallocate the associated memory.

Garbage Collection Algorithms

Object creation is done by the code the developer writes, the frameworks used, and their corresponding features. The developer is not required to deallocate the memory or dereference the objects. This is done automatically at the JVM level by the garbage collector. Since java's inception, there have been many updates to the algorithms which run behind the scene to free the memory. This section examines how they work.

Mark and Sweep

1. Stages of the Algorithm: Mark-and-sweep is a very basic algorithm that runs in two stages:
 - a. Marking live objects – this finds out all the objects that are still alive.
 - b. Removing Unreachable Objects – This gets rid of everything else, the supposedly dead and unused objects.
2. Identifying the Garbage Collection Roots: To start with, GC defines some specific objects are *garbage collection roots*, e.g., these include local variables and input parameters of the currently executing methods, active threads, static fields of the loaded classes, and JNI references.



3. Traversing the In-memory Object Graph: The GC traverses the whole object graph in memory, starting from those roots and following references from the roots to other objects. Every object that the GC visits is marked as alive.
4. Stop-the-World Pause: The application threads need to be stopped for the marking to happen as the collector cannot really traverse the graph if it keeps on changing. This is called *stop-the-world pause*.
5. Object Memory Free-up Phase: The second stage is to get rid of unused objects to free up memory. This can be done in a variety of ways, as shown below.
6. Normal Deletion: Normal deletion removes unreferenced objects to free space and leaves referenced objects and pointers. The memory allocator – a kind of hashtable – holds references to blocks of free space where new object can be allocated. This is often referred to as *mark-sweep* allocation.
7. Deletion with Compacting: Only removing unused objects is not efficient, because blocks of free memory are scattered across the storage area and cause *OutOfMemoryError* if the created object is big such that no large enough memory block is available.
8. Motivation behind Compacting Assigned Objects: To solve the above issue, after deleting unreferenced objects, compacting is done on the remaining referenced objects. Here, compacting refers to the process of moving referenced objects together. This makes memory allocation easier, faster, and more reliable. This is often referred to as *mark-sweep-compact* algorithm.
9. Deletion with Copying: This is very similar to the mark and compact approach, as it too relocates all live objects. The important difference is that the target of relocation is a different memory region. This is often referred to as *mark-copy* algorithm.
10. Generational Java Memory Management Scheme: Much of the remaining analysis relies on the structures introduced in Java Memory Management, so it makes sense to review it. In particular, the concepts of young generation, old generation, and permanent generation need to be understood in good detail.



Concurrent Mark-Sweep (CMS) Garbage Collection

1. CMS GC Multi-threaded Garbage Collection: CMS garbage collection is essentially an upgraded mark-and-sweep method. It scans heap memory using multiple threads. It was modified to take advantage of faster systems and has additional performance enhancements.
2. Customized Collection for Young/Old Generation: It attempts to minimize the pauses due to garbage collection by doing most of the garbage collection work *concurrently* with running application threads. It uses the parallel stop-the-world *mark-copy* algorithm in the young generation and the mostly concurrent *mark-sweep* algorithm in the old generation.
3. Invocation of the CMS GC: To invoke CMS GC, the JVM is launched using the argument below: `-XX:+UseConcMarkSweepGC`

CMS GC Optimization Options

Flag	Description
<code>XX: +UseCMSInitiating/OccupancyOnly</code>	Indicates that the invoker wants to solely use occupancy as a criterion for starting a CMS collection operation
<code>XX: CMSInitiating/OccupationFraction = 70</code>	Sets the percentage of CMS generation occupancy to start a CMS generation cycle
<code>XX: CMSTriggerRatio = 70</code>	This is the percentage of <i>MinHeapFreeRatio</i> in the CMS generation that is allocated prior to the start of a CMS cycle
<code>XX: CMSTriggerPermRatio = 90</code>	Sets the <i>MinHeapFreeRatio</i> in the CMS permanent generation that is allocated before starting a CMS collection cycle
<code>XX: CMSWaitDuration = 200</code>	This parameter specifies how long the CMS is allowed to wait for young generation collection
<code>XX: +UseParNewGC</code>	Elect to use the parallel algorithm for young generation collection



<i>XX: +CMSConcurrentMTEnabled</i>	Enables the use of multiple threads for concurrent phases
<i>XX: ConcGCThreads = 2</i>	Sets the number of parallel threads for concurrent phases
<i>XX: +CMSIncrementalMode</i>	Enables the incremental CMS (iCMS) mode
<i>XX: +CMSClassUnloadingEnabled</i>	If this is not enabled, CMS will not clean permanent space
<i>XX: +ExplicitGCInvokes/Concurrent</i>	This allows <i>System.gc()</i> to trigger concurrent collection instead of a full garbage collection cycle

Serial Garbage Collection

1. Mechanism of Serial Garbage Collection: This algorithm uses *mark-copy* for young generation and *mark-sweep-compact* for old generation. It works on a single thread. When executing, it freezes all other threads until the garbage collection operations have concluded.
2. Limitations of Serial Garbage Collection: Due to the thread freezing nature of serial garbage collection, it is only feasible for very small programs.
3. Invocation of the Serial Garbage Collector: The JVM arguments below are used to invoke the serial garbage collector: *-XX: +UseSerialGC*

Parallel Garbage Collection

1. Mechanism of Parallel Garbage Collection: Similar to serial GC, parallel garbage collection uses *mark-copy* in the young generation and *mark-sweep-compact* in the old generation. Multiple concurrent threads are used for the marking and the copying/compacting phases. The number of threads may also be configured using the *-XX: ParallelGCThreads = N* option.
2. Improvements achieved by Parallel GC: Parallel garbage collector is suitable on multi-core machines in cases where the primary goal is to increase throughput by



efficient usage of existing system resources. Using this approach, GC cycle times can be considerably reduced.

3. Default Collector - Parallel vs G1: Till Java 8, parallel GC was used as the default garbage collector. Java 9 onwards uses G1 as the default garbage collector on 32-bit and 64-bit server configurations.
4. Invoking the Parallel GC Scheme: The JVM arguments below are used to invoke the parallel garbage collector: `-XX:+UseParallelGC`

G1 Garbage Collection

1. G1 – Enhanced CMS Collector: The G1 – or Garbage First – garbage collector was available in Java 7 and is designated to be the long-term replacement for the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector.
2. Parallel Collection over Smaller Sub-regions: G1 garbage collection approach involves segmenting the memory heap into multiple small regions, typically 2048. Each region is marked as either young generation – further divided into Eden regions and Survivor regions – or old generation. This allows GC to avoid collecting the entire heap at one, and instead approach the problem incrementally. It means that only a subset of these regions is considered at a time.
3. Maintaining Collection Statistics over Sub-regions: G1 keeps track of the amount of live data each region contains. The information is used in determining the regions that contain the most garbage, so they are collected first. This is why the scheme is named *garbage-first collection*.
4. Tweaking Stop-the-World Durations: Just like other algorithms, unfortunately, the compacting operation takes place using the *stop-the-world* approach. However, specific performance goals can be set as per the design objectives.
5. No Guarantee in Duration Limitations: The pause duration can be configured, e.g., no more than 10 milliseconds in any given second. Garbage first GC will do its best to



meet this goal with high probability, but not with certainty, since hard real-time is not guaranteed due to OS level thread management.

5. Invoking the G1 Garbage Collector: In Java 7/8 machines, the JVM argument to invoke the G1 garbage collector is `-XX: +UseG1GC`

G1 Optimization Options

Flag	Description
<code>XX: G1HeapRegionSize = 16m</code>	Size of the Heap region. The value will be a power of 2 and can range from 1 MB to 32 MB. The goal is to have around 2048 regions based on the minimum Java heap size.
<code>XX: MaxGCPauseMillis = 200</code>	Sets the target value for the desired maximum pause time. The default value is 200 milliseconds. The specified value does not adapt to the heap size.
<code>XX: G1ReservePercent = 5</code>	This determines the minimum reserve in the heap.
<code>XX: G1ConfidencePercent = 5</code>	This is the confidence coefficient pause prediction heuristics.
<code>XX: GCPauseIntervalMillis = 5</code>	This is the pause interval time slice per MMU in milliseconds.

G1 Configuration Flags

Flag	Description
<code>-Xms2048m -Xmx3g</code>	Sets the initial and the maximum heap size – young space plus tenured space.
<code>XX: +DisableExplicitGC</code>	This will cause the JVM to ignore any <i>System.gc()</i> method invocation by an application.
<code>XX: +UseGCOverheadLimit</code>	This policy is used to limit the time spent in garbage collection before an <i>OutOfMemory</i> error is thrown.
<code>XX: GCTimeLimit = 95</code>	This limits the proportion of time spent in garbage collection before an <i>OutOfMemory</i> error is thrown.



<i>XX:GCHeapFreeLimit = 5</i>	This sets the minimum percentage of free space after a full garbage collection before an <i>OutOfMemory</i> error is thrown. This is used with <i>GCTimeLimit</i> .
<i>XX:InitialHeapSize = 3g</i>	Sets the initial heap size – young space plus tenured space.
<i>XX:MaxHeapSize = 3g</i>	Sets the maximum heap size – young space plus tenured space.
<i>XX:NewSize = 128m</i>	Sets the initial size of young space.
<i>XX:MaxNewSize = 128m</i>	Sets the maximum size of young space.
<i>XX:SurvivorRatio = 15</i>	Sets the size of the single survivor space as a portion of the Eden space size.
<i>XX:PermSize = 512m</i>	Sets the initial size of the permanent size.
<i>XX:MaxPermSize = 512m</i>	Sets the maximum size of the permanent size.
<i>-Xss512k</i>	Sets the size of the area dedicated to each thread in bytes.

G1 Logging Flags

Flag	Description
<i>-verbose:gc</i> OR <i>-XX:+PrintGC</i>	This prints the basic garbage collection information.
<i>-XX:+PrintGCDetails</i>	This will print more detailed garbage collection information.
<i>XX:+PrintGCTimestamps</i>	This prints timestamps for each garbage collection event. The seconds are sequential and begin from the JVM start time.
<i>XX:+PrintGCDateStamps</i>	This prints the date-stamps for each garbage collection event.
<i>-Xloggc:</i>	Using this directs the garbage collection output to a file instead of the console.
<i>XX:+Print\TenuringDistribution</i>	This prints detailed information regarding young space following each collection cycle.
<i>-XX:+PrintTLAB</i>	This flag prints TLAB collection statistics.
<i>-XX:+PrintReferenceGC</i>	This flag prints the time for reference processing (that is, weak, soft, and so on) during stop-the-world pauses.
<i>-XX:+HeapDump\OnOutOfMemoryError</i>	This flag creates heap dump file in an out-of-memory condition.



Processes and Threads

Overview

1. Java Concurrent Programming using Threads: In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.
2. Time Slicing Based on Concurrent Threads: A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus have only one thread executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called *time slicing*.
3. Concurrency on Single/Multiple Cores: It is becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads, but concurrency is possible even on simple systems, without multiple processors or execution cores.

Processes

1. Self-contained Unit of Execution: Each process has a self-contained execution environment. A process has a complete, private set of basic run-time resources; in particular, each process has its own memory space.
2. Mechanisms of Inter-Process-Communication: Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may actually be in fact a set of cooperating processes. To facilitate



communications between processes, most operating systems support *inter-process communication* (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes in the same system, but processes on different systems.

3. Process Creation Scheme in Java: Most implementations of the Java VM run as a single process. A Java application can create additional processes using a *ProcessBuilder* object.

Threads

1. Execution Environment Provided by Threads: Threads are sometimes called *lightweight* processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.
2. Resources Shared by Processes' Threads: Threads exist within a process – every process has at least one. Threads share the processes' resources, including memory and open files. This makes for efficient, but potentially problematic, communication.
3. Process' Main and Supporting Threads: Multi-threaded execution is an essential feature of the Java platform. Every application has at least one thread – or several, if one counts *system* threads that do things like memory management and signal handling. But, from the application programmer's point of view, one starts with just a single thread, called the *main thread*. This thread has the ability to create additional threads, as is demonstrated later.



Thread Objects

Overview

Each thread is associated with an instance of the class *Thread*. There are two basic strategies for using *Thread* objects to create a concurrent application:

- a. To directly control thread creation and management, simply instantiate *Thread* each time the application needs to initiate an asynchronous task.
- b. To abstract thread management from the rest of the application, the application's tasks are passed to an *executor*.

This chapter documents the use of *Thread* objects. Executors are discussed along with other high-level concurrency objects.

Defining and Starting a Thread

1. Provide a *Runnable* Object: An application that creates an instance of *Thread* must provide the code that will run in that method. There are two ways to do this: First, the *Runnable* interface defines a single method, *run*, meant to contain the code executed in the thread. The *Runnable* object is passed to the *Thread* constructor.
2. Subclass *Thread*: The *Thread* class itself implements *Runnable*, though its *run* method does nothing. An application can subclass *Thread*, providing its own implementation of *run*. Both the idioms above will need to invoke *Thread.invoke* in order to start a new thread.
3. Advantages of Implementing *Runnable* Interface: The first idiom, which employs a *Runnable* object, is more general, because the *Runnable* object can subclass a class other than *Thread*. The second idiom is easier to use in simple applications, but is



limited by the fact that the task class must be a descendant of *Thread*. This chapter focusses on the first approach, which separates the *Runnable* task from the *Thread* object that executes the task. Not only is this approach more flexible, it is also applicable to the high-level thread management APIs covered later.

4. Functionality Provided by *Thread* Class: The *Thread* class defines a number of methods useful for thread management. These include *static* methods, which provide information, or affect the status of, the thread invoking the method. The other methods are invoked from other threads involved in managing the thread and the *Thread* object. The next sections examine some of these methods.

Pausing Execution with *sleep*

1. Setting up the *sleep* Time: *Thread.sleep*: *Thread.sleep* causes the current thread to suspend execution for a specified period. This is an efficient means of making the processor threads available to the threads of an application or other applications that may be running on a computer system. The *sleep* method can be used for pacing, and waiting for another thread with duties that are understood to have time requirements.
2. Time Guarantees Provided by *sleep*: Two overloaded versions of *sleep* are provided: one specifies the sleep time to the millisecond and one specifies the sleep time to the nanosecond. However, these sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS. Also, the sleep period can be terminated by interrupts as seen in a later section. In any case, one cannot assume that invoking *sleep* will suspend the thread for precisely the time period specified.
3. Handling *InterruptedException* Thrown in *sleep*: The method calling *sleep* declares that it *throws InterruptedException*. This is an exception that *sleep* throws when another thread interrupts the current thread while *sleep* is active. If an application has not defined another thread to cause an interrupt, it need not bother catching *InterruptedException*.



Interrupts

1. Purpose behind the Interrupt Construct: An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It is up to the developer to decide exactly how a thread responds to an interrupt, but it is common for the thread to terminate. This is the usage emphasized in this section.
2. Mechanism for Sending an Interrupt: A thread sends an interrupt by invoking *interrupt* on the *Thread* object of the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Supporting Interruption

1. Thread Supporting its own Interruption: How does a thread support its own interruption? This depends on what it is currently doing. If the thread is frequently invoking methods that throw *InterruptedException*, it simply returns from the *run* method after it catches the exception. Many methods that throw *InterruptedException*, such as *sleep*, are designed to cancel their current operation and return immediately when an interrupt is received.
2. Methods that don't throw *InterruptedException*: What if a thread goes a long time without invoking a method that throw *InterruptedException*? Then it must periodically invoke *Thread.interrupted*, which return *true* if an interrupt has been received.
3. Explicitly Checking for Triggered Interrupts: In simple applications, the code can simply test for a triggered interrupt and exit the thread if one has been received. In more complex applications, it might make more sense to throw an



InterruptedException. This would allow interrupt handling code to be centralized in a *catch* clause.

The Interrupt Status Flag

1. Usage Caveats Underlying Interrupt Mechanism: The interrupt mechanism is implemented using an internal flag known as *interrupt status*. Invoking *Thread.interrupt* sets this flag. When a thread checks for an interrupt by invoking the static method *Thread.interrupted*, the interrupt status is cleared. The non-static *isInterrupted* method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.
2. Clearing Interrupt Status after throwing *InterruptedException*: By convention, any method that exits by throwing an *InterruptedException* clears the interrupt status when it does so. However, it is always possible that the interrupt status will be immediately set again, by another thread invoking *interrupt*.

Joins

The *join* method allows one thread to wait for the completion of another. If *t* is a *Thread* object whose thread is currently executing, *t.join()* causes the current thread to pause execution until *t*'s thread terminates. Overloads of *join* allow a developer to specify a waiting period. However, as with *sleep*, *join* is dependent on the OS for timing, so one cannot assume that *join* will wait exactly as long as specified. Further, like *sleep*, *join* responds to an interrupt by exiting with an *InterruptedException*.



Kubernetes

Overview

1. Focus of the Kubernetes System: *Kubernetes* is an open-source container orchestration system for automating computer application deployment, scaling, and management (Garrison (2016), Wikipedia (2020)).
2. Objectives of the System Design: It aims to provide a platform for automating deployment, scaling, and operations of application containers across clusters of hosts. It works with a range of container tools, including Docker.
3. Vendor Service Deployments using Kubernetes: Many closed services offer a Kubernetes-based platform or infrastructure as a service (PaaS or IaaS) on which Kubernetes can be deployed as a platform providing service. Many vendors also provide their own branded Kubernetes distributions.

Kubernetes Objects

1. Purpose behind the Kubernetes Objects: Kubernetes maintains a set of building blocks or primitives which collectively provide mechanisms that deploy, maintain, and scale applications based on CPU, memory, or custom metrics (Sharma (2017)).
2. Structure and Extensibility of Kubernetes: Kubernetes is loosely coupled and extensible to meet different workloads. The extensibility is provided in large part by the Kubernetes API, which is used by internal components as well as extensions and containers that run on Kubernetes. The platform exerts its control over compute and storage resources by defining resources as Objects, which can then be managed as such.



Pods

1. What is a Kubernetes Pod? A *pod* is a higher level of abstraction grouping containerized components. A pod consists of one or more containers that are guaranteed to be co-located on the host machine and can share resources. The basic scheduling unit in Kubernetes is a *pod*.
2. IP Addresses for Kubernetes Pods: Each pod in Kubernetes is assigned a unique *Pod IP address* within the cluster, which allows applications to use ports without the risk of conflict (Langemak (2015a)). Within the pod, all containers can reference each other on localhost, but a container within one pod has no way of directly addressing another container within another pod; for that, it has to use the pod IP address.
3. Name Service to Reference Pods: An application developer should never use the pod IP address though, to reference/invoke a capability in another pod, as pod IP addresses are ephemeral – the specific pod that they are addressing may be assigned to another pod IP address on re-start. Instead, they should use a reference to a Service, which holds a reference to the target pod at the specific pod IP address.
4. Local/Network Disk Volume Pods: A pod can define a Volume, such as a local disk directory or a network disk, and expose it to the containers in the pod (Strachan (2015a)). Pods can be managed manually through the Kubernetes API, or their management can be delegated to a controller.
5. Storage for ConfigMaps and Secrets: Such volumes are also the basis for Kubernetes features of ConfigMaps – which provide access to configuration through the file-system visible to the container – and Secrets – which provide access to credentials required to access remote resources securely by providing those credentials on the filesystem visible only to authorized containers.

ReplicaSets



1. Purpose of the ReplicaSet: A ReplicaSet's purpose is to maintain a stable set of replica pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical pods.
2. Selector Based ReplicaSet Specification: The ReplicaSets can also be said to be a grouping mechanism that lets Kubernetes maintain the number of instances that have been declared for a given pod. The definition of ReplicaSet uses a selector whose evaluation will result in identifying all pods that are associated with it.

Services

1. Definition of a Kubernetes Service: A Kubernetes source is a set of pods that work together, such as one-tier of a multi-tier application. The set of pods that constitute a service is defined by a label selector.
2. Modes of Service Discovery in Kubernetes: Kubernetes provides two modes of service discovery, using environmental variables of Kubernetes DNS. Service discovery assigns a stable IP address and DNS name to the service, and load balances traffic in a round-robin manner to network connections of that IP address among the pods matching the selector – even as failures cause the pods to move from machine to machine.
3. Exposing Front/Back End Services: By default, a service is exposed inside a cluster, e.g., back end pods might be grouped into a service, with requests from the front-end pods load balanced among them, but a service can also be exposed outside a cluster, e.g., for clients to reach front-end pods (Langemak (2015b)).

Volumes

1. Transient Nature of Kubernetes Filesystems: Filesystems in Kubernetes containers provide ephemeral storage, by default. This means that re-start of the pod will wipe



out any data on such containers, and therefore, this form of storage is quite limiting in anything but trivial applications.

2. Persistent Storage Mechanism - Kubernetes Volume: A Kubernetes Volume provides persistent storage that exists for the life of the pod itself. This storage can also be used as shared disk space for containers within the pod.
3. Restrictions on the Mount Points: Volumes are mounted at specific mount points within the container, which are defined by the pod configuration, and cannot mount onto other Volumes or link to other Volumes. The same Volume can be mounted at different points in the filesystem tree by different containers.

Namespaces

Kubernetes provides a partitioning of the resources it manages into non-overlapping sets called namespaces. They are intended for use in environments with many users spread across multiple teams or projects, even separating environments like development, test, and production.

ConfigMaps and Secrets

1. Types of Kubernetes Configuration Data: A common application challenge is deciding where to store and manage configuration information, some of which may contain sensitive data. Configuration data can be anything as fine-grained as individual properties or coarse-grained information like entire configuration files or JSON/XML documents.
2. Access to Configuration Data: Kubernetes provides two closely related mechanisms to deal with this need – *configmaps* and *secrets*, both of which allow for configuration changes to be made without requiring an application rebuild. The data from



configmaps and secrets will be made available to every single application to which these objects have been bound via deployment.

StatefulSets

1. Kubernetes Stateless versus Stateful Scaling: It is very easy to address the scaling of stateless applications: one simply adds more running pods – which is something that Kubernetes does very well. Stateful workloads are much harder, because the state needs to be preserved if a pod is restarted, and if the application is scaled up or down the state may need to be redistributed.
2. Ordering of the Stateful Instances: Databases are an example of stateful workloads. When run in high-availability mode, many databases come with the notion of a primary instance and secondary instance(s). In this case, the notion of ordering the instances is important.
3. Uniqueness of the Stateful Instances: Other applications like Kafka distribute the data among their brokers – so that one broker is not the same as another. In this case, the notion of instance uniqueness is important.
4. Uniqueness and Ordering among Pod Instances: StatefulSets are controllers that are provided by Kubernetes that enforce the properties of uniqueness and ordering among the instances of a pod and can be used to run stateful applications.

DaemonSets

Normally, the location where the pods are run are determined by the algorithm implemented in the Kubernetes scheduler. The ability to do this kind of pod scheduling is implemented by the feature called DaemonSets.



Secrets

Secrets contain ssh keys, passwords, and OAuth tokens for the pod.

Managing Kubernetes Objects: Labels and Selectors

1. Label Selector Based Object Matches: Kubernetes enables clients – users or internal components – to attach keys called *labels* to any API Object in the system, such as pods or nodes. Correspondingly, *label selectors* are queries against labels that resolve to matching objects.
2. Service Routing using Label Selectors: When a service is defined, one can define the label selector that will be used by the service router/load balancer to select the pod instances that the traffic will be routed to.
3. Controlling Service Routing Using Labels: Thus, simply changing the labels of the pods or changing the label selectors on the service can be used to control which pod gets the traffic and which doesn't, and this can be used to support various deployment patterns like blue-green deployments or A-B testing. This capability to dynamically control how services utilize implementing resources provides a loose coupling within the infrastructure.
4. Multi-Criterion Label Selection Example: For example, if the application's pods have labels for a system *tier* – with values such as *front_end*, *back_end*, for example – and a *release_track* – with values such as *canary*, *production*, for example – then an operation on all of *back_end* and *canary* nodes can use a label selector, such as:

$$tier = back_end \text{ AND } release_track = canary$$

Field Selectors



1. Object-Attribute-Value Based Selection: Just like fields, field selectors also let one select Kubernetes resources. Unlike labels, the selection is based on the attribute values inherent to the resource being selected, rather than user-defined categorization.
2. Built-in Metadata Field Selectors: *metadata.name* and *metadata.namespace* are field selectors that will be present on all Kubernetes objects. Other selectors can be used depending on the object/resource type.

Replication Controllers and Deployments

1. Focus of the Replication Controller: A ReplicaSet declares the number of instances of a pod that is needed, and the Replication Controller manages the system so that the number of healthy pods that are running matches the number of pods declared in the ReplicaSet, determined by evaluating its selector.
2. Kubernetes Deployments versus ReplicaSets: Deployments are a higher-level management mechanism for ReplicaSets. While the Replication Controller manages the scale of the ReplicaSet, Deployments will manage what happens to the ReplicaSet – whether an update has to be rolled out, rolled back, etc.
3. Deployment Induced Changes in ReplicaSets: When Deployments are scaled up or down, this results in the declaration of the ReplicaSet changing – and this change in the declared state is managed by the Replication Controller.

Cluster API

1. Programmatic Control of Kubernetes Clusters: The design principles underlying Kubernetes allow one to programmatically create, configure, and manage Kubernetes clusters. This function is exposed via an API called the Cluster API.
2. Cluster as a Controllable Resource: A key concept embodied in the API is the notion that the Kubernetes cluster itself is a resource object that can be managed just like any



other Kubernetes resources. Similarly, machines that make up the cluster are also treated as a Kubernetes resource.

3. Cloud Provider Specific API Functions: The API has two pieces – the core API, and a provider implementation. The provider implementation consists of cloud-provider specific functions that let Kubernetes provide the cluster API in a fashion that is well-integrated with the cloud provider's services and resources.

Architecture

Kubernetes follows a primary/replica architecture. The components of Kubernetes can be divided into those that manage an individual node, and those that are part of the control plane.

Kubernetes Control Plane

1. Kubernetes Master and Control Plane: The Kubernetes master is the main controlling unit of the cluster, managing the workload and directing communication across the system. The Kubernetes control plane consists of various components, each its own process, that can run both on a single master node or on multiple masters supporting high-availability clusters. The various components of the Kubernetes control plane are as follows.
2. etcd - Distributed Key-Value Store: etcd is a persistent, lightweight, distributed key-value store developed by CoreOS that reliably stores the configuration data of the cluster, representing the overall state of the cluster at any given point in time.
3. Choosing Cluster Consistency over Availability: Just like Apache ZooKeeper, etcd is a system that favors consistency over availability in the event of network partition. This consistency is crucial for correctly scheduling and operating services.



4. Usage of etcd's Watch API: The Kubernetes API Server uses etcd's Watch API to monitor the cluster or roll out critical configuration changes or simply restore any divergences of the state of the cluster back to what was declared by the deployer.
5. Example - Creating Additional Pod Instance: As an example, if the deployer specified that three instances of a particular pod need to be running, this fact is stored in etcd. If it is found that only two instances are running, this delta will be detected by comparison with the etcd data, and Kubernetes will use this to schedule the creation of an additional instance of that pod.
6. The Kubernetes API Server Component: The API Server is a key component and serves the Kubernetes API using JSON over HTTP, which provides both the internal and the external interfaces to Kubernetes (Marhubi (2015a)). The API server process and validates REST requests and updates the state of the API objects in etcd, thereby allowing clients to configure workloads and containers across worker nodes (Ellingwood (2018)).
7. Role of the Kubernetes Scheduler: The scheduler is the pluggable component that selects which node an unscheduled pod – the basic entity managed by the scheduler – runs on, based on resource availability. The scheduler tracks resource use on each node to ensure that the workload is not scheduled in excess of available resources.
8. Matching Resource Supply and Workload Demand: For this purpose, the scheduler must know the resource requirements, resource availability, and other user-provided constraints and policy directives such as quality-of-service, affinity/anti-affinity requirements, data locality, and so on. In essence, the scheduler's role is to match the resource *supply* to workload *demand*.
9. The Kubernetes Controller Manager Component: The controller is a reconciliation loop that drives the actual cluster state toward the desired cluster state, communicating with the API Server to create, update, and delete the resources it manages, i.e., the pods, the services, and the end-points (Marhubi (2015a)). The Controller Manager is a component that manages a set of core Kubernetes controllers.
10. The Kubernetes Replication Controller Component: One kind of controller is the Replication Controller, which handles replication and scaling by running a specified



number of pods across the cluster. It also handles creating replacement pods if the underlying node fails.

11. Kubernetes DaemonSet and Job Controllers: Other controllers that are a part of the core Kubernetes system include a DaemonSet Controller for running exactly one pod on every machine, and a Job Controller for running pods that run to completion, e.g., as part of a batch job (Sanders (2015)).
12. Control Exercised using Label Selection: The set of pods that a controller manages is determined by the label selection that are part of the controller's definition.

Kubernetes Node

1. Nodes where Containers are Deployed: A Node, also known as a Worker or a Minion, is a machine where containers/workloads are deployed. Every node in the cluster must run a container runtime such as Docker, as well as the below-mentioned components, for communication with the primary for network configuration of these containers.
2. Kubelet - Kubernetes Node Health Monitor: Kubelet is responsible for running the state of each node, ensuring that all containers in the node are healthy. It takes care of starting, stopping, and maintaining application containers organized into pods as directed by the control plane (Marhubi (2015b)).
3. Mechanism Employed in Monitoring the Node: Kubelet monitors the state of a pod, and if not in the desired state, the pod re-deploys to the same node. Node status is relayed every few seconds via heartbeat messages to the primary. Once the primary detects a node failure, the Replication Controller observes this state change and launches pods on other healthy nodes.
4. Role of the Kube-Proxy: The Kube-Proxy is an implementation of the network proxy and a load balancer, and it supports service abstraction along with other networking operations. It is responsible for routing the traffic to the appropriate container based on the IP and the port number of the incoming request.



5. Kubernetes Workload/Container Runtime: A container resides inside a pod. The container is the lowest level of a micro-service, which holds the running applications, libraries, and their dependencies. Containers can be exposed to the world through an external IP address. Kubernetes supports Docker containers since its first version, and in July 2016 rkt container engine was added.

Add-Ons

1. Additional Special Purpose Cluster Applications: Add-ons operate just like any other application running within the cluster: they are implemented via pods and services, and are only different in that they implement features of the Kubernetes cluster. The pods may be managed by Deployments, Replication Controllers, and so on. There are many add-ons, and the list is growing. Some of the more important are as follows.
2. DNS: All Kubernetes clusters should have a cluster DNS; it is a mandatory feature. Cluster DNS is a DNS server, in addition to the other DNS server(s) in the environment, which serves DNS records for Kubernetes services. Containers started by Kubernetes automatically include this DNS server in their DNS searches.
3. Web UI: This is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.
4. Container Resource Monitoring: Providing a reliable application runtime, and being able to scale it up or down in response to workloads, means being able to continuously and effectively monitor workload performance. Container Resource Monitoring provides this capability by recording metrics about the containers in a central database, and provides UI for browsing that data. The cAdvisor is a component of the slave node that provides a limited metric monitoring capability. There are full metrics pipelines as well, such as Prometheus, which can meet most monitoring needs.



5. Cluster-Level Logging: Logs should have a separate storage and lifecycle independent of nodes, pods, or containers. Otherwise, node or pod failures can cause loss of event data. The ability to do this is called cluster-level logging, and such mechanisms are responsible for saving container logs to a central log-store with searching/browsing interface. Kubernetes provides no native storage for log data, but one can integrate many existing logging solutions into the Kubernetes cluster.

Microservices

Kubernetes is commonly used as a way to host microservice-based implementation, because it and its associated ecosystem of tools provides all the capabilities needed to address key concerns of any microservice architecture.

Kubernetes Persistence Architecture

1. Containers – Enabling Portability of Software: Containers emerged as a way to make software portable. The container contains all the packages you need to run a service. The provided filesystem makes the containers extremely portable and easy to use in development. A container can also be moved from development to test or production with no or relatively few changes.
2. Reasons for Requiring Container Persistence: Historically, Kubernetes was suitable only for stateless services. However, many applications have a database, which require persistence, which leads to creation of persistence storage for Kubernetes. Implementing persistent storages is one of the top challenges for Kubernetes administrators, DevOps, and cloud engineers. Containers may be more ephemeral, but more and more of their data is not, so one needs to ensure data's survival in case of container termination or hardware failure.



3. Persistent Storage for Containerized Applications: When deploying containers with Kubernetes or containerized applications, companies often realize that they need persistent storage. They need to provide fast and reliable storage for databases, root images, and other data used by the containers.
4. CNCF Kubernetes Persistent Storage Guidelines: In addition to the landscape, the Cloud Native Computing Foundation (CNCF) has published other information about Kubernetes Persistent Storage including a blog helping to define the container attached storage pattern. This pattern can be thought of as one that uses Kubernetes itself as a component of the storage system or service.
5. Public Domain Storage Orchestration Projects: More information about the relative popularity of these and the other approaches can be found on the CNCF's landscape survey as well, which showed that OpenEBS from MayaData and Rook – a storage orchestration project – were the two projects most likely to be in evaluation.

References

- Ellingwood, J. (2018): [An Introduction to Kubernetes](#)
- Garrison, J. (2016): [Why Kubernetes is Abbreviated k8s](#)
- Langemak, J. (2015a): [Kubernetes 101 – Networking](#)
- Langemak, J. (2015b): [Kubernetes 101 – External Access into the Cluster](#)
- Marhubi, K. (2015a): [Kubernetes from the Ground Up: the API Server](#)
- Marhubi, K. (2015b): [What even is a kubelet?](#)
- Sanders, J. (2015): [Kubernetes: Exciting Experimental Features](#)
- Sharma, P. (2017): [Autoscaling based on CPU/Memory in Kubernetes — Part II](#)
- Strachan, J. (2015): [Kubernetes for Developers](#)
- Wikipedia (2020): [Kubernetes](#)



Apache Kafka

Overview

1. Stream-Processing Low-Latency Platform: *Apache Kafka* is an open-source stream-processing software developed by Apache Software Foundation, written in Java and Scala. The project aims to provide a unified high-throughput, low-latency platform for handling real-time data feeds.
2. Data Interfacing and Streams Library: Kafka can connect to external systems for data import/export via Kafka Connect and provides Kafka Streams, a Java Streams processing library (Wikipedia (2020)).
3. TCP Based Bulk Message Abstraction: Kafka uses a binary TCP protocol that is optimized for efficiency and relies on a *message set* abstraction that naturally groups messages together to reduce the overhead of network round trips. This leads to larger network packets, larger sequential disk operations, and contiguous memory blocks which allow Kafka to convert a bursty stream of random message writes into linear writes.

Architecture

1. Kafka Producers, Partitions, and Topics: Kafka stores key-value messages that come from arbitrarily many processors called *producers*. The data is partitioned into different *partitions* within different *topics*.
2. Ordering of Messages within Partitions: Within a partition, messages are strictly ordered by offsets – the position of message within a partition – and indexed and stored together with a timestamp. Other processors called *consumers* can read messages from partitions.



3. Streams Processing API and Interfaces: For streams processing, Kafka offers the Streams API that allows writing Java applications that consumer data from Kafka and writes results back to Kafka. Apache Kafka also works with external stream processing systems such as Apache Apex, Apache Flink, Apache Spark, Apache Storm, and Apache NiFi.
4. Kafka Brokers and Partition Distribution: Kafka runs on a cluster of one or more servers called brokers, and the partitions of all topics are distributed across the cluster nodes. Additionally, partitions are replicated across multiple brokers.
5. Stream Volumes and Transactional Features: This architecture allows Kafka to deliver massive streams of messages in a fault-tolerant fashion and has allowed it to replace some of the conventional messaging systems like java Messaging Service (JMS), Advanced Message Queue Protocol (AMQP), etc. Since the 0.11.0.0 release, Kafka offers *transactional writes* which provide exactly-once stream-processing using the Streams API.
6. Regular Topics - Purpose and Usage: Kafka supports two types of topics – regular and compacted. Regular topics can be configured with a retention time of space bound. If there are records that are older than the specified retention time or if the space bound is exceeded for a partition, Kafka is allowed to delete old data to free storage space. By default, topics are configured with a retention time of 7 days, but it is also possible to store data indefinitely.
7. Compacted Topics - Purpose and Usage: For compacted topics, records don't expire based on time or space bounds. Instead, Kafka treats the later messages as updates to the older messages with the same key and guarantees never to delete the latest message per key. Users can delete messages entirely by writing a so-called tombstone message with NULL value for a specific key. To this purpose, there are five major APIs in Kafka:
8. Producer API: Permits an application to publish streams of records.
9. Consumer API: Permits an application to subscribe to topics and processes streams of records.



10. Connector API: Executes the reusable producer and consumer APIs that can link the topics to existing applications.
11. Streams API: This API converts the input streams to output and produces the result.
12. Admin API: This API is used to manage Kafka topics, brokers, and other objects.
13. Message Protocol and API/Usage: The consumer and the producer APIs build on top of the Kafka messaging protocols and offer a reference implementation for Kafka consumer and producer clients in Java. The underlying message protocol is a binary protocol that developers can use to write their own consumer or producer clients in any programming language. This unlocks Kafka from the JVM eco-system.

Connect API

1. Handling Data from External Systems: Kafka Connect – or Connect API – is a framework to import/export data from/to other systems. It was added in Kafka 0.9.0.0 release and uses the Producer and the Consumer APIs internally. The Connect Framework itself executes the so-called *connectors* that implement the actual logic to read/write data from other systems.
2. API for Implementing Custom Connectors: The Connect API defines the programming interface that must be implemented to build a custom connector. Many open source and commercial connectors for popular data systems are available already. However, Kafka itself does not include any production ready connection.

Streams API

1. Java Based Streams Processing Library: Kafka Streams – or Streams API – is a stream-processing library written in Java. The library allows for the development of stateful stream-processing applications that are scalable, elastic, and fully fault-tolerant.



2. DSL Standard and Custom Operators: The main API is a stream-processing domain-specific language – DSL – that offers high-level operations like filter, map, grouping, windowing, aggregation, joins and the notion of tables. Additionally, the Processor API can be used to implement custom operators for a more low-level development approach. The DSL and the Processor API can be mixed, too.
3. Stateful Stream Processing using RocksDB: For stateful Streams processing, Kafka Streams uses RocksDB to maintain local operator state. Because RocksDB can write to disk, the maintained state can be larger than the available main memory. For fault-tolerance, all updates to the local state stores are also written into a topic in the Kafka cluster. This allows re-creating the state by reading those topics and feeding all data into RocksDB.

Version Compatibility

Up to version 0.9.x, Kafka brokers are backward compatible with older clients only. Since Kafka 0.10.0.0, brokers are also forward compatible with newer clients. If a newer client connects to an older broker, it can only use the features that the broker supports. For the Streams API, full compatibility starts with version 0.10.0.0; a 0.10.0.0 Kafka Streams application is not compatible with 0.10.0.0 or older nodes.

Performance

Monitoring end-to-end performance requires tracking metrics from brokers, consumers, and producers. In addition to monitoring ZooKeeper, which Kafka uses for coordination among consumers (Mouzakitis (2016)). There are currently several platforms to track Kafka performance, either open-source, like LinkedIn's Burrow (<https://github.com/linkedin/Burrow/wiki>), or paid, like DataDog and Deep BI. In



addition to these platforms, collecting Kafka data can also be performed using tools commonly bundled with Java, including JConsole.

References

- Mouzakitis, E. (2016): [Monitoring Kafka Performance Metrics](#)
- Wikipedia (2020): [Apache Kafka](#)