Kubernetes Replication Controller, Replica Set and Deployments: Understanding replication options

As a container management tool, Kubernetes was designed to orchestrate multiple containers and replication, and in fact there are currently several ways to do it. In this article, we’ll look at three options: Replication Controllers, Replica Sets, and Deployments.

What is Kubernetes replication for?

Before we go into how you would do replication, let’s talk about why.  Typically you would want to replicate your containers (and thereby your applications) for several reasons, including:

* **Reliability:** By having multiple versions of an application, you prevent problems if one or more fails.  This is particularly true if the system replaces any containers that fail.
* **Load balancing:** Having multiple versions of a container enables you to easily send traffic to different instances to prevent overloading of a single instance or node. This is something that Kubernetes does out of the box, making it extremely convenient.
* **Scaling:** When load does become too much for the number of existing instances, Kubernetes enables you to easily scale up your application, adding additional instances as needed.

Replication is appropriate for numerous use cases, including:

* **Microservices-based applications:** In these cases, multiple small applications provide very specific functionality.
* **Cloud native applications:** Because cloud-native applications are based on the theory that any component can fail at any time, replication is a perfect environment for implementing them, as multiple instances are baked into the architecture.
* **Mobile applications:** Mobile applications can often be architected so that the mobile client interacts with an isolated version of the server application.

Kubernetes has multiple ways in which you can implement replication.

Types of Kubernetes replication

In this article, we’ll discuss three different forms of replication: the Replication Controller, Replica Sets, and Deployments.

Replication Controller

The Replication Controller is the original form of replication in Kubernetes.  It’s being replaced by Replica Sets, but it’s still in wide use, so it’s worth understanding what it is and how it works.

A Replication Controller is a structure that enables you to easily create multiple pods, then make sure that that number of pods always exists. If a pod does crash, the Replication Controller replaces it.

Replication Controllers also provide other benefits, such as the ability to scale the number of pods, and to update or delete multiple pods with a single command.

You can create a Replication Controller with an imperative command, or declaratively, from a file.  For example, create a new file called rc.yaml and add the following text:

apiVersion: v1

kind: ReplicationController

metadata:

 name: soaktestrc

spec:

 replicas: 3

 selector:

   app: soaktestrc

 template:

   metadata:

     name: soaktestrc

     labels:

       app: soaktestrc

   spec:

     containers:

     - name: soaktestrc

       image: nickchase/soaktest

       ports:

       - containerPort: 80

Most of this structure should look familiar from our [discussion of Deployments](https://www.mirantis.com/blog/introduction-to-yaml-creating-a-kubernetes-deployment/); we’ve got the name of the actual Replication Controller (soaktestrc) and we’re designating that we should have 3 replicas, each of which are defined by the template.  The selector defines how we know which pods belong to this Replication Controller.

Now tell Kubernetes to create the Replication Controller based on that file:

# **kubectl create -f rc.yaml**

replicationcontroller "soaktestrc" created

Let’s take a look at what we have using the describe command:

# **kubectl describe rc soaktestrc**

Name:           soaktestrc

Namespace:      default

Image(s):       nickchase/soaktest

Selector:       app=soaktestrc

Labels:         app=soaktestrc

Replicas:       3 current / 3 desired

Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed

No volumes.

Events:

 FirstSeen     LastSeen        Count   From                            SubobjectPath   Type   Reason                   Message

 ---------     --------        -----   ----                            -------------   --------------                  -------

 1m            1m              1       {replication-controller }                       Normal SuccessfulCreate Created pod: soaktestrc-g5snq

 1m            1m              1       {replication-controller }                       Normal SuccessfulCreate Created pod: soaktestrc-cws05

 1m            1m              1       {replication-controller }                       Normal SuccessfulCreate Created pod: soaktestrc-ro2bl

As you can see, we’ve got the Replication Controller, and there are 3 replicas, of the 3 that we wanted.  All 3 of them are currently running.  You can also see the individual pods listed underneath, along with their names.  If you ask Kubernetes to show you the pods, you can see those same names show up:

# **kubectl get pods**

NAME               READY     STATUS    RESTARTS   AGE

soaktestrc-cws05   1/1       Running   0          3m

soaktestrc-g5snq   1/1       Running   0          3m

soaktestrc-ro2bl   1/1       Running   0          3m

Next we’ll look at Replica Sets, but first let’s clean up:

# **kubectl delete rc soaktestrc**

replicationcontroller "soaktestrc" deleted

# **kubectl get pods**

As you can see, when you delete the Replication Controller, you also delete all of the pods that it created.

Replica Sets

Replica Sets are a sort of hybrid, in that they are in some ways more powerful than Replication Controllers, and in others they are less powerful.

Replica Sets are declared in essentially the same way as Replication Controllers, except that they have more options for the selector.  For example, we could create a Replica Set like this:

apiVersion: extensions/v1beta1

kind: ReplicaSet

metadata:

 name: soaktestrs

spec:

 replicas: 3

 selector:

   matchLabels:

     app: soaktestrs

 template:

   metadata:

     labels:

       app: soaktestrs

  environment: dev

   spec:

     containers:

     - name: soaktestrs

       image: nickchase/soaktest

       ports:

       - containerPort: 80

In this case, it’s more or less the same as when we were creating the Replication Controller, except we’re using matchLabels instead of label.  But we could just as easily have said:

...

spec:

 replicas: 3

 selector:

**matchExpressions:**

**- {key: app, operator: In, values: [soaktestrs, soaktestrs, soaktest]}**

**- {key: teir, operator: NotIn, values: [production]}**

 template:

   metadata:

...

In this case, we’re looking at two different conditions:

1. The app label must be soaktestrc, soaktestrs, or soaktest
2. The tier label (if it exists) must not be production

Let’s go ahead and create the Replica Set and get a look at it:

# **kubectl create -f replicaset.yaml**

replicaset "soaktestrs" created

# **kubectl describe rs soaktestrs**

Name:           soaktestrs

Namespace:      default

Image(s):       nickchase/soaktest

Selector:       app in (soaktest,soaktestrs),teir notin (production)

Labels:         app=soaktestrs

Replicas:       3 current / 3 desired

Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed

No volumes.

Events:

 FirstSeen     LastSeen        Count   From                            SubobjectPath   Type    Reason                   Message

 ---------     --------        -----   ----                            -------------   --------------                   -------

 1m            1m              1       {replicaset-controller }                        Normal  SuccessfulCreate Created pod: soaktestrs-it2hf

 1m            1m              1       {replicaset-controller }                       Normal  SuccessfulCreate Created pod: soaktestrs-kimmm

 1m            1m              1       {replicaset-controller }                        Normal  SuccessfulCreate Created pod: soaktestrs-8i4ra

# **kubectl get pods**

NAME               READY     STATUS    RESTARTS   AGE

soaktestrs-8i4ra   1/1       Running   0          1m

soaktestrs-it2hf   1/1       Running   0          1m

soaktestrs-kimmm   1/1       Running   0          1m

As you can see, the output is pretty much the same as for a Replication Controller (except for the selector), and for most intents and purposes, they are similar.  The major difference is that the rolling-update command works with Replication Controllers, but won’t work with a Replica Set.  This is because Replica Sets are meant to be used as the backend for Deployments.

Let’s clean up before we move on.

# **kubectl delete rs soaktestrs**

replicaset "soaktestrs" deleted

# **kubectl get pods**

Again, the pods that were created are deleted when we delete the Replica Set.

Deployments

Deployments are intended to replace Replication Controllers.  They provide the same replication functions (through Replica Sets) and also the ability to rollout changes and roll them back if necessary.

Let’s create a simple Deployment using the same image we’ve been using.  First create a new file, deployment.yaml, and add the following:

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

 name: soaktest

spec:

 replicas: 5

 template:

   metadata:

     labels:

       app: soaktest

   spec:

     containers:

     - name: soaktest

       image: nickchase/soaktest

       ports:

       - containerPort: 80

Now go ahead and create the Deployment:

# **kubectl create -f deployment.yaml**

deployment "soaktest" created

Now let’s go ahead and describe the Deployment:

# **kubectl describe deployment soaktest**

Name:                   soaktest

Namespace:              default

CreationTimestamp:      Sun, 05 Mar 2017 16:21:19 +0000

Labels:                 app=soaktest

Selector:               app=soaktest

Replicas:               5 updated | 5 total | 5 available | 0 unavailable

StrategyType:           RollingUpdate

MinReadySeconds:        0

RollingUpdateStrategy:  1 max unavailable, 1 max surge

OldReplicaSets:         <none>

NewReplicaSet:          soaktest-3914185155 (5/5 replicas created)

Events:

 FirstSeen     LastSeen        Count   From                            SubobjectPath   Type    Reason                   Message

 ---------     --------        -----   ----                            -------------   --------------                   -------

 38s           38s             1       {deployment-controller }                        Normal  ScalingReplicaSet        Scaled up replica set soaktest-3914185155 to 3

 36s           36s             1       {deployment-controller }                        Normal  ScalingReplicaSet        Scaled up replica set soaktest-3914185155 to 5

As you can see, rather than listing the individual pods, Kubernetes shows us the Replica Set.  Notice that the name of the Replica Set is the Deployment name and a hash value.

A complete discussion of updates is out of scope for this article — we’ll cover it in the future — but couple of interesting things here:

* The StrategyType is RollingUpdate. This value can also be set to Recreate.
* By default we have a minReadySeconds value of 0; we can change that value if we want pods to be up and running for a certain amount of time — say, to load resources — before they’re truly considered “ready”.
* The RollingUpdateStrategy shows that we have a limit of 1 maxUnavailable — meaning that when we’re updating the Deployment, we can have up to 1 missing pod before it’s replaced, and 1 maxSurge, meaning we can have one extra pod as we scale the new pods back up.

As you can see, the Deployment is backed, in this case, by Replica Set soaktest-3914185155. If we go ahead and look at the list of actual pods…

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3914185155-7gyja   1/1       Running   0          2m

soaktest-3914185155-lrm20   1/1       Running   0          2m

soaktest-3914185155-o28px   1/1       Running   0          2m

soaktest-3914185155-ojzn8   1/1       Running   0          2m

soaktest-3914185155-r2pt7   1/1       Running   0          2m

… you can see that their names consist of the Replica Set name and an additional identifier.

Passing environment information: identifying a specific pod

Before we look at the different ways that we can affect replicas, let’s set up our deployment so that we can see what pod we’re actually hitting with a particular request.  To do that, the image we’ve been using displays the pod name when it outputs:

<?php

$limit = $\_GET['limit'];

if (!isset($limit)) $limit = 250;

for ($i; $i < $limit; $i++){

    $d = tan(atan(tan(atan(tan(atan(tan(atan(tan(atan(123456789.123456789))))))))));

}

**echo "Pod ".$\_SERVER['POD\_NAME']." has finished!\n";**

?>

As you can see, we’re displaying an environment variable, POD\_NAME.  Since each container is essentially it’s own server, this will display the name of the pod when we execute the PHP.

Now we just have to pass that information to the pod.

We do that through the use of the Kubernetes Downward API, which lets us pass environment variables into the containers:

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

 name: soaktest

spec:

 replicas: 3

 template:

   metadata:

     labels:

       app: soaktest

   spec:

     containers:

     - name: soaktest

       image: nickchase/soaktest

       ports:

       - containerPort: 80

**env:**

**- name: POD\_NAME**

**valueFrom:**

**fieldRef:**

**fieldPath: metadata.name**

As you can see, we’re passing an environment variable and assigning it a value from the Deployment’s metadata.  (You can find more information on metadata here.)

So let’s go ahead and clean up the Deployment we created earlier…

# **kubectl delete deployment soaktest**

deployment "soaktest" deleted

# **kubectl get pods**

… and recreate it with the new definition:

# **kubectl create -f deployment.yaml**

deployment "soaktest" created

Next let’s go ahead and expose the pods to outside network requests so we can call the nginx server that is inside the containers:

# **kubectl expose deployment soaktest --port=80 --target-port=80 --type=NodePort**

service "soaktest" exposed

Now let’s describe the services we just created so we can find out what port the Deployment is listening on:

# **kubectl describe services soaktest**

Name:                   soaktest

Namespace:              default

Labels:                 app=soaktest

Selector:               app=soaktest

Type:                   NodePort

IP:                     11.1.32.105

Port:                   <unset> 80/TCP

NodePort:               <unset> **30800**/TCP

Endpoints:              10.200.18.2:80,10.200.18.3:80,10.200.18.4:80 + 2 more...

Session Affinity:       None

No events.

As you can see, the NodePort is 30800 in this case; in your case it will be different, so make sure to check.  That means that each of the servers involved is listening on port 30800, and requests are being forwarded to port 80 of the containers.  That means we can call the PHP script with:

http://[HOST\_NAME OR HOST\_IP]:[PROVIDED PORT]

In my case, I’ve set the IP for my Kubernetes hosts to hostnames to make my life easier, and the PHP file is the default for nginx, so I can simply call:

# **curl http://kube-2:30800**

Pod soaktest-3869910569-xnfme has finished!

So as you can see, this time the request was served by pod soaktest-3869910569-xnfme.

Recovering from crashes: Creating a fixed number of replicas

Now that we know everything is running, let’s take a look at some replication use cases.

The first thing we think of when it comes to replication is recovering from crashes. If there are 5 (or 50, or 500) copies of an application running, and one or more crashes, it’s not a catastrophe.  Kubernetes improves the situation further by ensuring that if a pod goes down, it’s replaced.

Let’s see this in action.  Start by refreshing our memory about the pods we’ve got running:

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-qqwqc   1/1       Running   0          11m

soaktest-3869910569-qu8k7   1/1       Running   0          11m

soaktest-3869910569-uzjxu   1/1       Running   0          11m

soaktest-3869910569-x6vmp   1/1       Running   0          11m

soaktest-3869910569-xnfme   1/1       Running   0          11m

If we repeatedly call the Deployment, we can see that we get different pods on a random basis:

# **curl http://kube-2:30800**

Pod soaktest-3869910569-xnfme has finished!

# **curl http://kube-2:30800**

Pod soaktest-3869910569-x6vmp has finished!

# **curl http://kube-2:30800**

Pod soaktest-3869910569-uzjxu has finished!

# **curl http://kube-2:30800**

Pod soaktest-3869910569-x6vmp has finished!

# **curl http://kube-2:30800**

Pod soaktest-3869910569-uzjxu has finished!

# **curl http://kube-2:30800**

Pod soaktest-3869910569-qu8k7 has finished!

To simulate a pod crashing, let’s go ahead and delete one:

# **kubectl delete pod soaktest-3869910569-x6vmp**

pod "soaktest-3869910569-x6vmp" deleted

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-516kx   1/1       Running   0          18s

soaktest-3869910569-qqwqc   1/1       Running   0          27m

soaktest-3869910569-qu8k7   1/1       Running   0          27m

soaktest-3869910569-uzjxu   1/1       Running   0          27m

soaktest-3869910569-xnfme   1/1       Running   0          27m

As you can see, pod \*x6vmp is gone, and it’s been replaced by \*516kx.  (You can easily find the new pod by looking at the AGE column.)

If we once again call the Deployment, we can (eventually) see the new pod:

# **curl http://kube-2:30800**

Pod soaktest-3869910569-516kx has finished!

Now let’s look at changing the number of pods.

Scaling up or down: Manually changing the number of replicas

One common task is to scale up a Deployment in response to additional load. Kubernetes has autoscaling, but we’ll talk about that in another article.  For now, let’s look at how to do this task manually.

The most straightforward way is to simply use the scale command:

# **kubectl scale --replicas=7 deployment/soaktest**

deployment "soaktest" scaled

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-2w8i6   1/1       Running   0          6s

soaktest-3869910569-516kx   1/1       Running   0          11m

soaktest-3869910569-qqwqc   1/1       Running   0          39m

soaktest-3869910569-qu8k7   1/1       Running   0          39m

soaktest-3869910569-uzjxu   1/1       Running   0          39m

soaktest-3869910569-xnfme   1/1       Running   0          39m

soaktest-3869910569-z4rx9   1/1       Running   0          6s

In this case, we specify a new number of replicas, and Kubernetes adds enough to bring it to the desired level, as you can see.

One thing to keep in mind is that Kubernetes isn’t going to scale the Deployment down to be below the level at which you first started it up.  For example, if we try to scale back down to 4…

# **kubectl scale --replicas=4 -f deployment.yaml**

deployment "soaktest" scaled

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-l5wx8   1/1       Running   0          11s

soaktest-3869910569-qqwqc   1/1       Running   0          40m

soaktest-3869910569-qu8k7   1/1       Running   0          40m

soaktest-3869910569-uzjxu   1/1       Running   0          40m

soaktest-3869910569-xnfme   1/1       Running   0          40m

… Kubernetes only brings us back down to 5, because that’s what was specified by the original deployment.

Deploying a new version: Replacing replicas by changing their label

Another way you can use deployments is to make use of the selector.  In other words, if a Deployment controls all the pods with a tier value of dev, changing a pod’s teir label to prod will remove it from the Deployment’s sphere of influence.

This mechanism enables you to selectively replace individual pods. For example, you might move pods from a dev environment to a production environment, or you might do a manual rolling update, updating the image, then removing some fraction of pods from the Deployment; when they’re replaced, it will be with the new image. If you’re happy with the changes, you can then replace the rest of the pods.

Let’s see this in action.  As you recall, this is our Deployment:

# **kubectl describe deployment soaktest**

Name:                   soaktest

Namespace:              default

CreationTimestamp:      Sun, 05 Mar 2017 19:31:04 +0000

Labels:                 app=soaktest

Selector:               app=soaktest

Replicas:               3 updated | 3 total | 3 available | 0 unavailable

StrategyType:           RollingUpdate

MinReadySeconds:        0

RollingUpdateStrategy:  1 max unavailable, 1 max surge

OldReplicaSets:         <none>

NewReplicaSet:         **soaktest-3869910569** (3/3 replicas created)

Events:

 FirstSeen     LastSeen        Count   From                            SubobjectPath   Type              Reason                  Message

 ---------     --------        -----   ----                            -------------   --------  ------                  -------

 50s           50s             1       {deployment-controller }                        Normal            ScalingReplicaSet       Scaled up replica set soaktest-3869910569 to 3

And these are our pods:

# **kubectl describe replicaset soaktest-3869910569**

Name:           soaktest-3869910569

Namespace:      default

Image(s):       nickchase/soaktest

Selector:       app=soaktest,pod-template-hash=3869910569

Labels:         app=soaktest

               pod-template-hash=3869910569

Replicas:       5 current / 5 desired

Pods Status:    5 Running / 0 Waiting / 0 Succeeded / 0 Failed

No volumes.

Events:

 FirstSeen     LastSeen        Count   From                            SubobjectPath   Type              Reason                  Message

 ---------     --------        -----   ----                            -------------   --------  ------                  -------

 2m            2m              1       {replicaset-controller }                        Normal            SuccessfulCreate        Created pod: soaktest-3869910569-0577c

 2m            2m              1       {replicaset-controller }                        Normal            SuccessfulCreate        Created pod: soaktest-3869910569-wje85

 2m            2m              1       {replicaset-controller }                        Normal            SuccessfulCreate        Created pod: soaktest-3869910569-xuhwl

 1m            1m              1       {replicaset-controller }                        Normal            SuccessfulCreate        Created pod: soaktest-3869910569-8cbo2

 1m            1m              1       {replicaset-controller }                        Normal            SuccessfulCreate        Created pod: soaktest-3869910569-pwlm4

We can also get a list of pods by label:

# **kubectl get pods -l app=soaktest**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-0577c   1/1       Running   0          7m

soaktest-3869910569-8cbo2   1/1       Running   0          6m

soaktest-3869910569-pwlm4   1/1       Running   0          6m

soaktest-3869910569-wje85   1/1       Running   0          7m

soaktest-3869910569-xuhwl   1/1       Running   0          7m

So those are our original soaktest pods; what if we wanted to add a new label?  We can do that on the command line:

# **kubectl label pods soaktest-3869910569-xuhwl experimental=true**

pod "soaktest-3869910569-xuhwl" labeled

# **kubectl get pods -l experimental=true**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-xuhwl   1/1       Running   0          14m

So now we have one experimental pod.  But since the experimental label has nothing to do with the selector for the Deployment, it doesn’t affect anything.

So what if we change the value of the app label, which the Deployment **is** looking at?

# **kubectl label pods soaktest-3869910569-wje85 app=notsoaktest --overwrite**

pod "soaktest-3869910569-wje85" labeled

In this case, we need to use the overwrite flag because the app label already exists. Now let’s look at the existing pods.

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-0577c   1/1       Running   0          17m

soaktest-3869910569-4cedq   1/1       Running   0          4s

soaktest-3869910569-8cbo2   1/1       Running   0          16m

soaktest-3869910569-pwlm4   1/1       Running   0          16m

soaktest-3869910569-wje85   1/1       Running   0          17m

soaktest-3869910569-xuhwl   1/1       Running   0          17m

As you can see, we now have six pods instead of five, with a new pod having been created to replace \*wje85, which was removed from the deployment. We can see the changes by requesting pods by label:

# **kubectl get pods -l app=soaktest**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-0577c   1/1       Running   0          17m

soaktest-3869910569-4cedq   1/1       Running   0          20s

soaktest-3869910569-8cbo2   1/1       Running   0          16m

soaktest-3869910569-pwlm4   1/1       Running   0          16m

soaktest-3869910569-xuhwl   1/1       Running   0          17m

Now, there is one wrinkle that you have to take into account; because we’ve removed this pod from the Deployment, the Deployment no longer manages it.  So if we were to delete the Deployment…

# **kubectl delete deployment soaktest**

deployment "soaktest" deleted

The pod remains:

# **kubectl get pods**

NAME                        READY     STATUS    RESTARTS   AGE

soaktest-3869910569-wje85   1/1       Running   0          19m

You can also easily replace all of the pods in a Deployment using the –all flag, as in:

# **kubectl label pods --all app=notsoaktesteither --overwrite**

But remember that you’ll have to delete them all manually!