ContainerNotifier with Probe

Table of Contents

Motivation

In order to protect Kubernetes stateful workloads, we want to take application snapshots and backups. To ensure application consistency, we need a mechanism to send a command to the pod to quiesce an application before taking a snapshot of persistent volume(s) and un-quiesce afterwards. There are also other use cases that require the user to send a signal to the Pod to trigger certain commands. See this [issue](#) for more information. Following are some of those use cases:

- Run a command in a pod to quiesce an application before taking a snapshot and un-quiesce it after the snapshot is taken to ensure application consistency. (quiesce/freeze unquiesce/unfreeze/thaw)
- Send a signal to a pod for reloading configurations.
- Send a signal to flush logs, change log verbosity, etc.
- Build-in notifications, i.e., Eviction.

Since there are multiple use cases that require a way to trigger a command to run in a pod, we are thinking about a solution that is more general and secure than the typical CRD approach. Having Kubelet execute the hooks instead of an external controller would be more secure as an external controller is considerably easier to be compromised than kubelet and be able to arbitrarily exec on any pod.

Proposal

In phase 1, this API proposal adds an inline pod definition for sending commands to containers and an API object to send a request to trigger those commands.

An external controller such as the application snapshot controller would signal when a command needs to run by creating a request notification API object. A SINGLE *trusted* controller will be implemented to watch the objects and run the command when it is created and update its status accordingly.

In phase 2, this API proposal adds an inline pod definition for container signals and an API object to send a request to trigger those signals.

Logic in the SINGLE *trusted* controller will be moved to Kubelet. Kubelet watches the objects and runs the command when it is created and updates its status accordingly.

In phase 3, a Probe may be added if needed as an inline pod definition to verify whether the signal is delivered or whether the command is run and results in the desired outcome.

API Changes

Phase 1

There are two parts in the API changes:

- Add an inline pod definition for ContainerNotifierAction - defines a command to run in the container.

- Add a Notification API object - this is a request for the container notifier.

Phase 2

Move controller logic to Kubelet.
Add signal to the inline pod definition:

- In the inline pod definition for ContainerNotifierAction, allows a signal to be defined and sent to the container.

Phase 3

- If needed, add an inline pod definition for Probe - probes the container to verify that ContainerNotifierAction has brought the container to the desired state.

Inline Pod Definition for ContainerNotifier

Add []ContainerNotifier to the Container struct:

```
type Container struct {
    …….
    // +optional
    Lifecycle *Lifecycle `json:"lifecycle,omitempty" protobuf:"bytes,12,opt,name=lifecycle"`

   // Notifiers can be triggered by Notification resources.  What they mean are
   // up to the Pod owner to define, but we may choose to define some commonly
   // used names.  Each notifier must have a unique name within the container
   // definition, but the same name may be used in different containers of the
   // same pod.  When a Notification of a given name as a container notifier is created
   // (and the pod to which the container belongs to is selected by the Notification's
   // spec.podSelector), the notifier will run in the container.
   // +optional
   Notifiers []ContainerNotifier
   …….
}

type ContainerNotifier struct {
  // ~~name must be a DNS_LABEL,~~ Name must be unique within a container.
  // Names are label-style, with an optional prefix. Names without the prefix
  // such as `quiesce` are reserved for Kubernetes-defined "well-known" names.
  // Names with a prefix such as `example.com/label-style` are custom names.
  // This refers to a ContainerNotifierAction.
  // Kubelet executes the command defined in ContainerNotifierAction.
  Name string
```

```
  Action *ContainerNotifierAction
}

// ContainerNotifier describes a "signal" that can be sent from outside the
// container.
type ContainerNotifierAction struct {
    // handler describes how this notifier is delivered to the container.
    Handler ContainerNotifierHandler

    // Number of seconds after which the notifier times out.
    // Defaults to 1 second. Minimum value is 1.
    // +optional
    TimeoutSeconds int32
}


// Handler defines a specific action that should be taken
type ContainerNotifierHandler struct {
    // Exec specifies the action to take.
    // +optional
    Exec *ExecAction `json:"exec,omitempty" protobuf:"bytes,1,opt,name=exec"`
}
```

Definition for ExecAction is here for reference.

Notification API Object

The Notification object is a request for a ContainerNotifier. It will be an in-tree API object in the "node.k8s.io" API group so that Kubelet can monitor and trigger the actions. The external controller creates the Notification object to request a ContainerNotifier.
All containers in a selected pod which have a "ContainerNotifier" defined in their spec with the "ContainerNotifierName" will have their corresponding "ContainerNotifierAction" executed. Moreover, there is no guarantee of ordering on execution.

```
type Notification struct {
        metav1.TypeMeta
        // +optional
        metav1.ObjectMeta

        // Spec defines the behavior of a notification.
```

```
	// +optional
	Spec NotificationSpec

	// Status defines the current state of a notification.
	// +optional
	Status NotificationStatus
}

type NotificationSpec struct {
	// This contains the name of the ContainerNotifier to send to a container.
	ContainerNotifierName string

	 // PodSelector specifies a label query over a set of pods.
	 // +optional
	 PodSelector *metav1.LabelSelector

	// ExpirationSeconds is the duration of the Notification object.
	// As the Notification object approaches expiration, Kubelet will start
	// deleting the Notification objects that are in completed status.
	// This is for garbage collection of Notification objects.
	// Defaults to 1 hour and must be at least 10 minutes.
	// +optional
	ExpirationSeconds int32

}
```

NotificationStatus

A PodNotificationStatus represents the current state of a Notification for a specific pod. A PodNotificationStatus contains ContainerNotificationStatuses of all the containers which have the corresponding ContainerNotifier specified.
PodNotificationStatus is created when a Notifier starts to run in a pod.

A PodNotificationStatus is considered to be successful only if all of its ContainerNotificationStatuses are successful. A Notification is considered to be successful only if all PodNotificationStatuses are successful.

Kubelet will **NOT** retry if any Notifier didn't run successfully. The external controller can request Notification again to do retries.

```
// NotificationStatus represents the current state of a Notification
type NotificationStatus struct {
	// This is a list of PodNotificationStatus, with each status representing information
	// about how Notification is executed in containers inside a pod.
	// +optional
	PodNotificationStatuses []PodNotificationStatus
}

// PodNotificationStatus represents the current state of a Notification for a specific pod
type PodNotificationStatus struct {
	// This field is required
	PodName string

	// This field is required
	ContainerNotificationStatuses []ContainerNotificationStatus
 }

// ContainerNotificationStatus represents the current state of a Notification for a specific
container
// The “Succeed” parameter is false until it completes successfully when it will be changed to
true.
// If an error occurs, the “Error” parameter will be set.
type ContainerNotificationStatus struct {
	// This field is required
	ContainerName string

	 // If not set, it is nil, indicating Action has not started
	 // If set, it means Action has started at the specified time
	// +optional
	Timestamp *int64

	// ActionSucceed is set to true when the action is executed in the container successfully.
	// It will be set to false if the action cannot be executed successfully after
	//ActionTimeoutSeconds passes.
	// +optional
	Succeed *bool

	// The last error encountered when executing the action. The controller might update
	// this field each time
	// it retries the execution.
	// +optional
	Error *ActionError
```

```
}


type ActionError struct {
	// Type of the error
	// This is required
	ErrorType ErrorType

	// Error message
	// +optional
	Message *string

	// More detailed reason why error happens
	// +optional
	Reason *string

	// It indicates when the error occurred
	// +optional
	Timestamp *int64
}

type ErrorType string

// More error types could be added, e.g., Forbidden, Unauthorized, etc.
const (
	// The Notification times out
	Timeout ErrorType = "Timeout"

	// The Notification fails with an error
	Error ErrorType = "Error"
)
```

External controller deletes the Notification API objects after it is done.

~~Type NotificationStatus struct {~~
~~	Conditions []NotificationConditions~~
~~}~~

~~type NotificationConditionType string~~

~~// These are valid conditions of Notifications~~

```
// Kubelet sets these conditions accordingly
const (
    // Notification is triggered when Kubelet starts to run the command
    NotificationTriggered NotificationConditionType = "NotificationTriggered"
    // Notification is completed when the command finishes running
    // NotificationSucceeded should be true if it completes successfully
    // or false if it fails
    NotificationSucceeded NotificationConditionType = "NotificationSucceeded"
)

type NotificationCondition struct {
    Type NotificationConditionType
    // ConditionStatus can be “True”, “False”, or “Unknown”
    Status ConditionStatus
    // +optional
    LastProbeTime metav1.Time
    // +optional
    LastTransitionTime metav1.Time
    // +optional
    Reason string
    // +optional
    Message string
}
```

External controller deletes the Notification API objects after it is done.

New NotificationStatus

NotificationStatus represents the current state of a Notification. It has a list of ContainerNotificationStatus.
ContainerNotificationStatus represents the current state of a Notification for a specific container in a pod.
ContainerNotificationStatus is created when a Notifier starts to run in a container. The “Succeed” parameter is false until it completes successfully when it will be changed to true.
If an error occurs, the “Error” parameter will be set.
If not all ContainerNotificationStatus is successful, a Notification is not considered successful.
Retries will only happen in containers whose Notifier didn’t run successfully.

```
// NotificationStatus represents the current state of a Notification
type NotificationStatus struct {
```

```
    // This is a list of ContainerNotificationStatus, with each status representing information
    // about how Notification is executed in a container, including pod name, container name,
    // ActionTimestamp, ActionSucceed, etc.
    // +optional
    ContainerNotificationStatuses []ContainerNotificationStatus
}

// ContainerNotificationStatus represents the current state of a Notification for a specific
container in a pod
type ContainerNotificationStatus struct {
    // This field is required
    PodName string

    // This field is required
    ContainerName string

    // If not set, it is nil, indicating Action has not started
    // If set, it means Action has started at the specified time
    // +optional
    Timestamp *int64

    // ActionSucceed is set to true when the action is executed in the container successfully.
    // It will be set to false if the action cannot be executed successfully after
    //ActionTimeoutSeconds passes.
    // +optional
    Succeed *bool

    // The last error encountered when executing the action. The controller might update
    // this field each time
    // it retries the execution.
    // +optional
    Error *ActionError
}

type ActionError struct {
    // Type of the error
    // This is required
    ErrorType ErrorType

    // Error message
    // +optional
    Message *string
```

~~   // More detailed reason why error happens~~
~~   // +optional~~
~~   Reason *string~~
~~~~
~~    // It indicates when the error occurred~~
~~    // +optional~~
~~    Timestamp *int64~~
~~}~~

~~type ErrorType string~~

~~// More error types could be added, e.g., Forbidden, Unauthorized, etc.~~
~~const (~~
~~    // The Notification times out~~
~~    Timeout ErrorType = "Timeout"~~

~~    // The Notification fails with an error~~
~~    Error ErrorType = "Error"~~
~~)~~

## Phase 2 API Additions

In Phase 2, add a “signal” field to ContainerNotifierHandler.

```
// Handler defines a specific action that should be taken or a signal that should be delivered
// Only one of Exec and Signal should be set and not both
type ContainerNotifierHandler struct {
    // Exec specifies the action to take.
    // +optional
    Exec *ExecAction `json:"exec,omitempty" protobuf:"bytes,1,opt,name=exec"`

    // Signal specifies a signal to send to the container
    // +optional
    // define constants for signals?
    // validate the signals are valid?  windows?
++    Signal string
}
```
~~~~

Phase 3 API Additions

If needed, we may add a Probe to ContainerNotifier in Phase 3, to verify that the ContainerNotifierAction has actually delivered the signal or has run the command.

```
type ContainerNotifier struct {
   // Name must be unique within a container.
   // Names are label-style, with an optional prefix. Names without the prefix
   // such as `quiesce` are reserved for Kubernetes-defined "well-known" names.
   // Names with a prefix such as `example.com/label-style` are custom names.
-   // This refers to a ContainerNotifierAction and a Probe.
+   // This refers to a ContainerNotifierAction and a Probe.
+   // Kubelet executes the command defined in ContainerNotifierAction, and then
+   // calls the command defined in Probe to verify whether the action has
+   // resulted in a desired state.
+   // For example, if the ContainerNotifierAction has a command to quiesce the application,
+   // then the Probe has a command to verify that the application is indeed
+   // being quiesced. Kubelet will run the command to do a quiesce and wait for the quiescent
+   // probe to return success.
   Name string
   Action *ContainerNotifierAction
+   // Add a Probe in Phase 2. Probe defined in the core API here will be used.
+   Probe *Probe
}
```

Note: *In the above proposal, the Probe is highly-linked with the Action which only triggers when a notification is used (more complex, but more bounded attacks). An alternative solution would be to introduce a standalone feature that happens to work well with notifications (solving the scale and DOS issues).*

Kubelet Impact in Phase 2

When moving the logic from the SINGLE *trusted* controller to Kubelet, there will be impact on Kubelet.

Kubelet watches new Notification resources cluster-wide.

Kubelet must execute Notification command/signal against containers. CRI changes are required to support signals.

Kubelet must update Notification status (possible QPS issues trying to update for 100+ of pods).

Kubelet must have some method of measuring the success/failure of the Notification.
For exec, it depends on the return value of the ExecInContainer method.
For signals, it depends on the new CRI changes. If we modify the existing “Exec” Runtime interface to handle signals, we can also depend on the return value of the ExecInContainer method to determine whether it is successful or not.

Kubelet will not retry if the ContainerNotifier call fails. It is up to the external controller who requested the ContainerNotifier to send another request to retry.

CRI Changes

To support signals directly, we need to make changes in CRI. We can modify the existing “Exec” Runtime interface to add an additional input parameter “signal string” and handle a specified number of signals, starting with “SIGHUP”. For example, if the input parameter “signal” is “SIGHUP”, it is a signal to kill the container.

// Runtime is the interface to execute the commands and provide the streams.
type Runtime interface {
- Exec(containerID string, cmd []string, in io.Reader, out, err io.WriteCloser, tty bool, resize <-chan remotecommand.TerminalSize) error
+ Exec(containerID string, cmd []string, in io.Reader, out, err io.WriteCloser, tty bool, resize <-chan remotecommand.TerminalSize, signal string) error

Attach(containerID string, in io.Reader, out, err io.WriteCloser, tty bool, resize <-chan remotecommand.TerminalSize) error
PortForward(podSandboxID string, port int32, stream io.ReadWriteCloser) error
}

To support “SIGHUP” directly, we can either convert it to the command “kill -SIGHUP” and send it the same way as other commands, or we can translate that into a call to the docker client method ContainerKill.

https://github.com/moby/moby/blob/master/client/container_kill.go#L9

We can add a SignalKill method in ~/go/src/k8s.io/kubernetes/pkg/kubelet/dockershim/libdocker/kube_docker_client.go and call docker client ContainerKill, similar to how CreateExec method calls docker client ContainerExecCreate.

Implementation Plan

1. Phase 1 API definition will be added to Kubernetes. Implementation in this phase will not happen in Kubelet. Instead, it will be implemented in a single *trusted* controller which acts on Notifications via exec (with the goal being to move that into Kubelet in the next steps). In this phase, only exec will be included and signal will not be included as that involves changes to CRI.
2. Make CRI changes to support signal and move controller logic to Kubelet.
3. Add Probe if needed.

Example Workflows

Example Workflow with Quiesce Hooks

For example, there are 3 commands that need to run sequentially to quiesce before taking a snapshot of a mysql database. They are lockTables, flushDisk, and fsfreeze. After taking the snapshot, there are 2 commands to run sequentially to unquiesce. They are fsUnfreeze, unlockTables. For simplicity, assume we only need to run fsfreeze for one volume. These commands are defined in “Notifier []ContainerNotifier” inside the Container.

1. lockTables
External controller creates a Notification object to request the lockTables ContainerNotifier. Kubelet watches the Notification object and gets notified. It starts to run the lockTables command specified in the ContainerNotifier and sets NotificationTriggered status to True.

When the command finishes successfully, Kubelet sets the status of NotificationSucceeded to True.
If it fails, Kubelet retries until it times out. When that happens, it stops retrying and sets the NotificationSucceeded status to False.

2. flushDisk
If the lockTables command succeeds, the external controller will proceed to create a Notification object to request the flushDisk ContainerNotifier.

Kubelet starts to run the flushDisk command specified in the ContainerNotifier and sets NotificationTriggered status to True.

When the flushDisk command finishes successfully, Kubelet sets the status of NotificationSucceeded to True.
If it fails, Kubelet retries until it times out. When that happens, it stops retrying and sets the NotificationSucceeded status to False.

If the lockTables command fails, the external controller will create a Notification object to request unlockTables ContainerNotifier. It will not proceed to the next Notification and the snapshot creation will be marked as failure.

3. fsfreeze
If the flushDisk command succeeds, the external controller will create a Notification object to request the fsfreeze ContainerNotifier.

Kubelet starts to run the fsfreeze command specified in the ContainerNotifier and sets NotificationTriggered status to True.

When the fsfreeze command finishes successfully, Kubelet sets the status of NotificationSucceeded to True.
If it fails, Kubelet retries until it times out. When that happens, it stops retrying and sets the NotificationSucceeded status to False.

4. Take snapshot
If the fsfreeze command succeeds, the external controller will proceed to take a snapshot.

5. fsUnfreeze
After taking the snapshot, the external controller will create a Notification object to request the fsUnfreeze ContainerNotifier.
Even if the snapshot creation fails, the external controller will still create a Notification object to request fsUnfreeze.
If fsFreeze is called, fsUnfreeze should always be called.

Kubelet starts to run the fsUnfreeze command specified in the ContainerNotifier and sets NotificationTriggered status to True.

When the fsUnfreeze command finishes successfully, Kubelet sets the status of NotificationSucceeded to True.

If it fails, Kubelet retries until it times out. When that happens, it stops retrying and sets the NotificationSucceeded status to False.

6. unlockTables
If the fsUnfreeze command succeeds, the external controller will proceed to create a Notification object to request the unlockTables ContainerNotifier.
If lockTables is called, unlockTables should always be called.

Kubelet starts to run the unlockTables command specified in the ContainerNotifier and sets NotificationTriggered status to True.

When the unlockTables command finishes successfully, Kubelet sets the status of NotificationSucceeded to True.
If it fails, Kubelet retries until it times out. When that happens, it stops retrying and sets the NotificationSucceeded status to False.

7. It is the external controller's responsibility to make sure unquiesce is always called following a quiesce command for the snapshot use case. This means fsUnfreeze is always called after fsFreeze and unlockTables is always called after lockTables.

8. The external controller will delete all Notification objects it has created after the commands have completed.

Example Workflow with Sighup

For example, the user wants to send sighup to a container in a Pod. This signal is defined in the ContainterNotifierAction inside the Container.

External controller creates a Notification object to request the sighup ContainerNotifier.
Kubelet watches the Notification object and gets notified. It sends the sighup signal defined in the ContainerNotifierAction to the container. This is similar to "docker kill --signal=SIGHUP my_container".

Kubelet also sends a probe to check if the container is still running. If the probe detects that the container is stopped, Kubelet sets the Succeeded field in ContainerNotificationStatus to True.
If the container is still running, Kubelet will retry probes periodically until it times out. When that happens, it stops retrying and sets the ContainerNotificationStatus Succeeded field to False.

Example Workflow to Change Log Verbosity

For example, the user wants to change log level to verbose in a container in a Pod. This signal is defined in the ContainterNotifierAction inside the Container.

External controller creates a Notification object to request the ChangeLogLevel ContainerNotifier to change log to verbose.
Kubelet watches the Notification object and gets notified. It sends the ChangeLogLevel to verbose signal defined in the ContainerNotifierAction to the container. Kubelet also sends a probe to check the log level in the container.

If the probe detects that the log level inside the container is indeed verbose, Kubelet sets the Succeeded field in ContainerNotificationStatus to True.
If the probe detects that the log level is not changed yet, Kubelet will retry probes periodically until it times out. When that happens, it stops retrying and sets the ContainerNotificationStatus Succeeded field to False.

References

K8S Node Shutdown:
https://docs.google.com/document/d/1mPBLcNyrGzsLDA6unBn00mMwYzlP2tSct0n8lWfuRGE/edit

Original ContainerNotifier Proposal