Kubernetes 101

edj

Table of Contents

docs/kubeconfig-file
docs/kubectl
docs/labels
docs/logging
docs/man/kube-apiserver.1
docs/man/kube-controller-manager.1
docs/man/kube-proxy.1
docs/man/kube-scheduler.1
docs/man/kubelet.1
docs/man/README
docs/namespaces
docs/networking
docs/node
docs/overview
docs/ovs-networking
docs/pod-states
docs/pods
docs/README
docs/replication-controller
docs/resources
docs/roadmap
docs/salt
docs/services
docs/ui
docs/volumes
examples/cassandra/README
examples/guestbook/README
examples/guestbook-go/ src/README
examples/guestbook-go/README
examples/hazelcast/README
examples/kubernetes-namespaces/README
examples/mysql-wordpress-pd/README
examples/node-selection/README
examples/redis/README
examples/rethinkdb/README
examples/update-demo/README
walkthrough/k8s201
walkthrough/README

K8s Identity and Access Management Sketch

This document suggests a direction for identity and access management in the Kubernetes system.

Background

High level goals are: - Have a plan for how identity, authentication, and authorization will fit in to the API. - Have a plan for partitioning resources within a cluster between independent organizational units. - Ease integration with existing enterprise and hosted scenarios.

Actors

Each of these can act as normal users or attackers. - External Users: People who are accessing applications running on K8s (e.g. a web site served by webserver running in a container on K8s), but who do not have K8s API access. - K8s Users : People who access the K8s API (e.g. create K8s API objects like Pods) - K8s Project Admins: People who manage access for some K8s Users - K8s Cluster Admins: People who control the machines, networks, or binaries that comprise a K8s cluster. - K8s Admin means K8s Cluster Admins and K8s Project Admins taken together.

Threats

Both intentional attacks and accidental use of privilege are concerns.

For both cases it may be useful to think about these categories differently: - Application Path - attack by sending network messages from the internet to the IP/port of any application running on K8s. May exploit weakness in application or misconfiguration of K8s. - K8s API Path - attack by sending network messages to any K8s API endpoint. - Insider Path - attack on K8s system components. Attacker may have privileged access to networks, machines or K8s software and data. Software errors in K8s system components and administrator error are some types of threat in this category.

This document is primarily concerned with K8s API paths, and secondarily with Internal paths. The Application path also needs to be secure, but is not the focus of this document.

Assets to protect

External User assets: - Personal information like private messages, or images uploaded by External Users - web server logs

K8s User assets: - External User assets of each K8s User - things private to the K8s app, like: - credentials for accessing other services (docker private repos, storage services, facebook, etc) - SSL certificates for web servers - proprietary data and code

K8s Cluster assets: - Assets of each K8s User - Machine Certificates or secrets. - The value of K8s cluster computing resources (cpu, memory, etc).

This document is primarily about protecting K8s User assets and K8s cluster assets from other K8s Users and K8s Project and Cluster Admins.

Usage environments

Cluster in Small organization: - K8s Admins may be the same people as K8s Users. - few K8s Admins. - prefer ease of use to fine-grained access control/precise accounting, etc. - Product requirement that it be easy for potential K8s Cluster Admin to try out setting up a simple cluster.

Cluster in Large organization: - K8s Admins typically distinct people from K8s Users. May need to divide K8s Cluster Admin access by roles. - K8s Users need to be protected from each other. - Auditing of K8s User and K8s Admin actions important. - flexible accurate usage accounting and resource controls important. - Lots of automated access to APIs. - Need to integrate with existing enterprise directory, authentication, accounting, auditing, and security policy infrastructure.

Org-run cluster: - organization that runs K8s master components is same as the org that runs apps on K8s. - Minions may be on-premises VMs or physical machines; Cloud VMs; or a mix.

Hosted cluster: - Offering K8s API as a service, or offering a Paas or Saas built on K8s - May already offer web services, and need to integrate with existing customer account concept, and existing authentication, accounting, auditing, and security policy infrastructure. - May want to leverage K8s User accounts and accounting to manage their User accounts (not a priority to support this use case.) - Precise and accurate accounting of resources needed. Resource controls needed for hard limits (Users given limited slice of data) and soft limits (Users can grow up to some limit and then be expanded).

K8s ecosystem services: - There may be companies that want to offer their existing services (Build, CI, A/B-test, release automation, etc) for use with K8s. There should be some story for this case.

Pods configs should be largely portable between Org-run and hosted configurations.

Design

Related discussion: - https://github.com/GoogleCloudPlatform/kubernetes/issues/442 - https://github.com/GoogleCloudPlatform/kubernetes/issues/443

This doc describes two security profiles: - Simple profile: like single-user mode. Make it easy to evaluate K8s without lots of configuring accounts and policies. Protects from unauthorized users, but does not partition authorized users. - Enterprise profile: Provide mechanisms needed for large numbers of users. Defense in depth. Should integrate with existing enterprise security infrastructure.

K8s distribution should include templates of config, and documentation, for simple and enterprise profiles. System should be flexible enough for knowledgeable users to create intermediate profiles, but K8s developers should only reason about those two Profiles, not a matrix.

Features in this doc are divided into "Initial Feature", and "Improvements". Initial features would be candidates for version 1.00.

Identity

userAccount

K8s will have a `userAccount` API object. - `userAccount` has a UID which is immutable. This is used to associate users with objects and to record actions in audit logs. - `userAccount` has a name which is a string and human readable and unique among userAccounts. It is used to refer to users in Policies, to ensure that the Policies are human readable. It can be changed only when there are no Policy objects or other objects which refer to that name. An email address is a suggested format for this field. - `userAccount` is not related to the unix username of processes in Pods created by that userAccount. - `userAccount` API objects can have labels

The system may associate one or more Authentication Methods with a `userAccount` (but they are not formally part of the userAccount object.) In a simple deployment, the authentication method for a user might be an authentication token which is verified by a K8s server. In a more complex deployment, the authentication might be delegated to another system which is trusted by the K8s API to authenticate users, but where the authentication details are unknown to K8s.

Initial Features: - there is no superuser `userAccount` - `userAccount` objects are statically populated in the K8s API store by reading a config file. Only a K8s Cluster Admin can do this. - `userAccount` can have a default `namespace`. If API call does not specify a `namespace`, the default `namespace` for that caller is assumed. - `userAccount` is global. A single human with access to multiple namespaces is recommended to only have one userAccount.

Improvements: - Make `userAccount` part of a separate API group from core K8s objects like `pod`. Facilitates plugging in alternate Access Management.

Simple Profile: - single `userAccount`, used by all K8s Users and Project Admins. One access token shared by all.

Enterprise Profile: - every human user has own `userAccount`. - `userAccounts` have labels that indicate both membership in groups, and ability to act in certain roles. - each service using the API has own `userAccount` too. (e.g. `scheduler`, `repcontroller`) - automated jobs to denormalize the ldap group info into the local system list of users into the K8s userAccount file.

Unix accounts

A `userAccount` is not a Unix user account. The fact that a pod is started by a `userAccount` does not mean that the processes in that pod's containers run as a Unix user with a corresponding name or identity.

Initially: - The unix accounts available in a container, and used by the processes running in a container are those that are provided by the combination of the base operating system and the Docker manifest. - Kubernetes doesn't enforce any relation between `userAccount` and unix accounts.

Improvements: - Kubelet allocates disjoint blocks of root-namespace uids for each container. This may provide some defense-in-depth against container escapes. (https://github.com/docker/docker/pull/4572) - requires docker to integrate user namespace support, and deciding what getpwnam() does for these uids. - any features that help users avoid use of privileged containers (https://github.com/GoogleCloudPlatform/kubernetes/issues/391)

Namespaces

K8s will have a have a `namespace` API object. It is similar to a Google Compute Engine `project`. It provides a namespace for objects created by a group of people co-operating together, preventing name collisions with non-cooperating groups. It also serves as a reference point for authorization policies.

Namespaces are described in namespace.html.

In the Enterprise Profile: - a `userAccount` may have permission to access several `namespaces`.

In the Simple Profile: - There is a single `namespace` used by the single user.

Namespaces versus userAccount vs Labels: - `userAccounts` are intended for audit logging (both name and UID should be logged), and to define who has access to `namespaces`. - `labels` (see docs/labels.html) should be used to distinguish pods, users, and other objects that cooperate towards a common goal but are different in some way, such as version, or responsibilities. - `namespaces` prevent name collisions between uncoordinated groups of people, and provide a place to attach common policies for co-operating groups of people.

Authentication

Goals for K8s authentication: - Include a built-in authentication system with no configuration required to use in single-user mode, and little configuration required to add several user accounts, and no https proxy required. - Allow for authentication to be handled by a system external to Kubernetes, to allow integration with existing to enterprise authorization systems. The kubernetes namespace itself should avoid taking contributions of multiple authorization schemes. Instead, a trusted proxy in front of the apiserver can be used to authenticate users. - For organizations whose security requirements only allow FIPS compliant implementations (e.g. apache) for authentication. - So the proxy can terminate SSL, and isolate the CA-signed certificate from less trusted, higher-touch APIserver. - For organizations that already have existing SaaS web services (e.g. storage, VMs) and want a common authentication portal. - Avoid mixing authentication and authorization, so that authorization policies be centrally managed, and to allow changes in authentication methods without affecting authorization code.

Initially: - Tokens used to authenticate a user. - Long lived tokens identify a particular `userAccount`. - Administrator utility generates tokens at cluster setup. - OAuth2.0 Bearer tokens protocol, http://tools.ietf.org/html/rfc6750 - No scopes for tokens. Authorization happens in the API server - Tokens dynamically generated by apiserver to identify pods which are making API calls. - Tokens checked in a module of the APIserver. - Authentication in apiserver can be disabled by flag, to allow testing without authorization enabled, and to allow use of an authenticating proxy. In this mode, a query parameter or header added by the proxy will identify the caller.

Improvements: - Refresh of tokens. - SSH keys to access inside containers.

To be considered for subsequent versions: - Fuller use of OAuth (http://tools.ietf.org/html/rfc6749) - Scoped tokens. - Tokens that are bound to the channel between the client and the api server - http://www.ietf.org/proceedings/90/slides/slides-90-uta-0.pdf - http://www.browserauth.net

Authorization

K8s authorization should: - Allow for a range of maturity levels, from single-user for those test driving the system, to integration with existing to enterprise authorization systems. - Allow for centralized management of users and policies. In some organizations, this will mean that the definition of users and access policies needs to reside on a system other than k8s and encompass other web services (such as a storage service). - Allow processes running in K8s Pods to take on identity, and to allow narrow scoping of permissions for those identities in order to limit damage from software faults. - Have Authorization Policies exposed as API objects so that a single config file can create or delete Pods, Controllers, Services, and the identities and policies for those Pods and Controllers. - Be separate as much as practical from Authentication, to allow Authentication methods to change over time and space, without impacting Authorization policies.

K8s will implement a relatively simple Attribute-Based Access Control model. The model will be described in more detail in a forthcoming document. The model will - Be less complex than XACML - Be easily recognizable to those familiar with Amazon IAM Policies. - Have a subset/aliases/defaults which allow it to be used in a way comfortable to those users more familiar with Role-Based Access Control.

Authorization policy is set by creating a set of Policy objects.

The API Server will be the Enforcement Point for Policy. For each API call that it receives, it will construct the Attributes needed to evaluate the policy (what user is making the call, what resource they are accessing, what they are trying to do that resource, etc) and pass those attributes to a Decision Point. The Decision Point code evaluates the Attributes against all the Policies and allows or denies the API call. The system will be modular enough that the Decision Point code can either be linked into the APIserver binary, or be another service that the apiserver calls for each Decision (with appropriate time-limited caching as needed for performance).

Policy objects may be applicable only to a single namespace or to all namespaces; K8s Project Admins would be able to create those as needed. Other Policy objects may be applicable to all namespaces; a K8s Cluster Admin might create those in order to authorize a new type of controller to be used by all namespaces, or to make a K8s User into a K8s Project Admin.)

Accounting

The API should have a `quota` concept (see https://github.com/GoogleCloudPlatform/kubernetes/issues/442). A quota object relates a namespace (and optionally a label selector) to a maximum quantity of resources that may be used (see resources.html).

Initially: - a `quota` object is immutable. - for hosted K8s systems that do billing, Project is recommended level for billing accounts. - Every object that consumes resources should have a `namespace` so that Resource usage stats are roll-up-able to `namespace`. - K8s Cluster Admin sets quota objects by writing a config file.

Improvements: - allow one namespace to charge the quota for one or more other namespaces. This would be controlled by a policy which allows changing a billing_namespace= label on an object. - allow quota to be set by namespace owners for (namespace x label) combinations (e.g. let "webserver" namespace use 100 cores, but to prevent accidents, don't allow "webserver" namespace and "instance=test" use more than 10 cores. - tools to help write consistent quota config files based on number of minions, historical namespace usages, QoS needs, etc. - way for K8s Cluster Admin to incrementally adjust Quota objects.

Simple profile: - a single `namespace` with infinite resource limits.

Enterprise profile: - multiple namespaces each with their own limits.

Issues: - need for locking or "eventual consistency" when multiple apiserver goroutines are accessing the object store and handling pod creations.

Audit Logging

API actions can be logged.

Initial implementation: - All API calls logged to nginx logs.

Improvements: - API server does logging instead. - Policies to drop logging for high rate trusted API calls, or by users performing audit or other sensitive functions.

Kubernetes Proposal - Namespaces

Related PR:

Topic	Link
Identifiers.html	https://github.com/GoogleCloudPlatform/kubernetes/pull/1216
Access.html	https://github.com/GoogleCloudPlatform/kubernetes/pull/891
Indexing	https://github.com/GoogleCloudPlatform/kubernetes/pull/1183
Cluster Subdivision	https://github.com/GoogleCloudPlatform/kubernetes/issues/442

Background

High level goals:

- Enable an easy-to-use mechanism to logically scope Kubernetes resources
- Ensure extension resources to Kubernetes can share the same logical scope as core Kubernetes resources
- Ensure it aligns with access control proposal
- Ensure system has log n scale with increasing numbers of scopes

Use cases

Actors:

1. k8s admin - administers a kubernetes cluster
2. k8s service - k8s daemon operates on behalf of another user (i.e. controller-manager)
3. k8s policy manager - enforces policies imposed on k8s cluster
4. k8s user - uses a kubernetes cluster to schedule pods

User stories:

1. Ability to set immutable namespace to k8s resources
2. Ability to list k8s resource scoped to a namespace
3. Restrict a namespace identifier to a DNS-compatible string to support compound naming conventions
4. Ability for a k8s policy manager to enforce a k8s user's access to a set of namespaces
5. Ability to set/unset a default namespace for use by kubecfg client
6. Ability for a k8s service to monitor resource changes across namespaces
7. Ability for a k8s service to list resources across namespaces

Proposed Design

Model Changes

Introduce a new attribute *Namespace* for each resource that must be scoped in a Kubernetes cluster.

A *Namespace* is a DNS compatible subdomain.

```
// TypeMeta is shared by all objects sent to, or returned from the client
type TypeMeta struct {
  Kind              string    `json:"kind,omitempty"`
  Uid               string    `json:"uid,omitempty"`
  CreationTimestamp util.Time `json:"creationTimestamp,omitempty"`
  SelfLink          string    `json:"selfLink,omitempty"`
  ResourceVersion   uint64    `json:"resourceVersion,omitempty"`
  APIVersion        string    `json:"apiVersion,omitempty"`
  Namespace         string    `json:"namespace,omitempty"`
  Name              string    `json:"name,omitempty"`
}
```

An identifier, *UID*, is unique across time and space intended to distinguish between historical occurences of similar entities.

A *Name* is unique within a given *Namespace* at a particular time, used in resource URLs; provided by clients at creation time and encouraged to be human friendly; intended to facilitate creation idempotence and space-uniqueness of singleton objects, distinguish distinct entities, and reference particular entities across operations.

As of this writing, the following resources MUST have a *Namespace* and *Name*

- pod
- service
- replicationController
- endpoint

A *policy* MAY be associated with a *Namespace*.

If a *policy* has an associated *Namespace*, the resource paths it enforces are scoped to a particular *Namespace*.

k8s API server

In support of namespace isolation, the Kubernetes API server will address resources by the following conventions:

The typical actors for the following requests are the k8s user or the k8s service.

Action	HTTP Verb	Path	Description
CREATE	POST	/api/{version}/ns/{ns}/{resourceType}/	Create instance of {resourceType} in namespace {ns}
GET	GET	/api/{version}/ns/{ns}/{resourceType}/{name}	Get instance of {resourceType} in namespace {ns} with {name}
UPDATE	PUT	/api/{version}/ns/{ns}/{resourceType}/{name}	Update instance of {resourceType} in namespace {ns} with {name}
DELETE	DELETE	/api/{version}/ns/{ns}/{resourceType}/{name}	Delete instance of {resourceType} in namespace {ns} with {name}
LIST	GET	/api/{version}/ns/{ns}/{resourceType}	List instances of {resourceType} in namespace {ns}
WATCH	GET	/api/{version}/watch/ns/{ns}/{resourceType}	Watch for changes to a {resourceType} in namespace {ns}

The typical actor for the following requests are the k8s service or k8s admin as enforced by k8s Policy.

Action	HTTP Verb	Path	Description
WATCH	GET	/api/{version}/watch/{resourceType}	Watch for changes to a {resourceType} across all namespaces
LIST	GET	/api/{version}/list/{resourceType}	List instances of {resourceType} across all namespaces

The legacy API patterns for k8s are an alias to interacting with the *default* namespace as follows.

Action	HTTP Verb	Path	Description
CREATE	POST	/api/{version}/{resourceType}/	Create instance of {resourceType} in namespace *default*
GET	GET	/api/{version}/{resourceType}/{name}	Get instance of {resourceType} in namespace *default*
UPDATE	PUT	/api/{version}/{resourceType}/{name}	Update instance of {resourceType} in namespace *default*
DELETE	DELETE	/api/{version}/{resourceType}/{name}	Delete instance of {resourceType} in namespace *default*

The k8s API server verifies the *Namespace* on resource creation matches the *{ns}* on the path.

The k8s API server will enable efficient mechanisms to filter model resources based on the *Namespace*. This may require the creation of an index on *Namespace* that could support query by namespace with optional label selectors.

The k8s API server will associate a resource with a *Namespace* if not populated by the end-user based on the *Namespace* context of the incoming request. If the *Namespace* of the resource being created, or updated does not match the *Namespace* on the request, then the k8s API server will reject the request.

TODO: Update to discuss k8s api server proxy patterns

k8s storage

A namespace provides a unique identifier space and therefore must be in the storage path of a resource.

In etcd, we want to continue to still support efficient WATCH across namespaces.

Resources that persist content in etcd will have storage paths as follows:

/registry/{resourceType}/{resource.Namespace}/{resource.Name}

This enables k8s service to WATCH /registry/{resourceType} for changes across namespace of a particular {resourceType}.

Upon scheduling a pod to a particular host, the pod's namespace must be in the key path as follows:

/host/{host}/pod/{pod.Namespace}/{pod.Name}

k8s Authorization service

This design assumes the existence of an authorization service that filters incoming requests to the k8s API Server in order to enforce user authorization to a particular k8s resource. It performs this action by associating the *subject* of a request with a *policy* to an associated HTTP path and verb. This design encodes the *namespace* in the resource path in order to enable external policy servers to function by resource path alone. If a request is made by an identity that is not allowed by policy to the resource, the request is terminated. Otherwise, it is forwarded to the apiserver.

k8s controller-manager

The controller-manager will provision pods in the same namespace as the associated replicationController.

k8s Kubelet

There is no major change to the kubelet introduced by this proposal.

kubecfg client

kubecfg supports following:

```
kubecfg [OPTIONS] ns {namespace}
```

To set a namespace to use across multiple operations:

```
$ kubecfg ns ns1
```

To view the current namespace:

```
$ kubecfg ns
Using namespace ns1
```

To reset to the default namespace:

```
$ kubecfg ns default
```

In addition, each kubecfg request may explicitly specify a namespace for the operation via the following OPTION

--ns

When loading resource files specified by the -c OPTION, the kubecfg client will ensure the namespace is set in the message body to match the client specified default.

If no default namespace is applied, the client will assume the following default namespace:

- default

The kubecfg client would store default namespace information in the same manner it caches authentication information today as a file on user's file system.

Kubernetes Proposal - Admission Control

Related PR:

Topic	Link
Separate validation from RESTStorage	https://github.com/GoogleCloudPlatform/kubernetes/issues/2977

Background

High level goals:

- Enable an easy-to-use mechanism to provide admission control to cluster
- Enable a provider to support multiple admission control strategies or author their own
- Ensure any rejected request can propagate errors back to the caller with why the request failed

Authorization via policy is focused on answering if a user is authorized to perform an action.

Admission Control is focused on if the system will accept an authorized action.

Kubernetes may choose to dismiss an authorized action based on any number of admission control strategies.

This proposal documents the basic design, and describes how any number of admission control plug-ins could be injected.

Implementation of specific admission control strategies are handled in separate documents.

kube-apiserver

The kube-apiserver takes the following OPTIONAL arguments to enable admission control

Option	Behavior
admission_control	Comma-delimited, ordered list of admission control choices to invoke prior to modifying or deleting an object.
admission_control_config_file	File with admission control configuration parameters to boot-strap plug-in.

An **AdmissionControl** plug-in is an implementation of the following interface:

```
package admission

// Attributes is an interface used by a plug-in to make an admission decision on a
type Attributes interface {
  GetNamespace() string
  GetKind() string
  GetOperation() string
  GetObject() runtime.Object
}

// Interface is an abstract, pluggable interface for Admission Control decisions.
type Interface interface {
  // Admit makes an admission decision based on the request attributes
  // An error is returned if it denies the request.
  Admit(a Attributes) (err error)
}
```

A **plug-in** must be compiled with the binary, and is registered as an available option by providing a name, and implementation of admission.Interface.

```
func init() {
  admission.RegisterPlugin("AlwaysDeny", func(client client.Interface, config io.R
}
```

Invocation of admission control is handled by the **APIServer** and not individual **RESTStorage** implementations.

This design assumes that **Issue 297** is adopted, and as a consequence, the general framework of the APIServer request/response flow will ensure the following:

1. Incoming request
2. Authenticate user
3. Authorize user
4. If operation=create|update, then validate(object)
5. If operation=create|update|delete, then admission.Admit(requestAttributes)

a. invoke each admission.Interface object in sequence

6. Object is persisted

If at any step, there is an error, the request is canceled.

Admission control plugin: LimitRanger

Background

This document proposes a system for enforcing min/max limits per resource as part of admission control.

Model Changes

A new resource, **LimitRange**, is introduced to enumerate min/max limits for a resource type scoped to a Kubernetes namespace.

```
const (
  // Limit that applies to all pods in a namespace
  LimitTypePod string = "Pod"
  // Limit that applies to all containers in a namespace
  LimitTypeContainer string = "Container"
)

// LimitRangeItem defines a min/max usage limit for any resource that matches on k
type LimitRangeItem struct {
  // Type of resource that this limit applies to
  Type string `json:"type,omitempty"`
  // Max usage constraints on this kind by resource name
  Max ResourceList `json:"max,omitempty"`
  // Min usage constraints on this kind by resource name
  Min ResourceList `json:"min,omitempty"`
}

// LimitRangeSpec defines a min/max usage limit for resources that match on kind
type LimitRangeSpec struct {
  // Limits is the list of LimitRangeItem objects that are enforced
  Limits []LimitRangeItem `json:"limits"`
}

// LimitRange sets resource usage limits for each kind of resource in a Namespace
type LimitRange struct {
  TypeMeta   `json:",inline"`
  ObjectMeta `json:"metadata,omitempty"`

  // Spec defines the limits enforced
  Spec LimitRangeSpec `json:"spec,omitempty"`
}

// LimitRangeList is a list of LimitRange items.
type LimitRangeList struct {
  TypeMeta `json:",inline"`
  ListMeta `json:"metadata,omitempty"`

  // Items is a list of LimitRange objects
  Items []LimitRange `json:"items"`
}
```

AdmissionControl plugin: LimitRanger

The **LimitRanger** plug-in introspects all incoming admission requests.

It makes decisions by evaluating the incoming object against all defined **LimitRange** objects in the request context namespace.

The following min/max limits are imposed:

Type: Container

ResourceName	Description
cpu	Min/Max amount of cpu per container
memory	Min/Max amount of memory per container

Type: Pod

ResourceName	Description
cpu	Min/Max amount of cpu per pod
memory	Min/Max amount of memory per pod

If the incoming object would cause a violation of the enumerated constraints, the request is denied with a set of messages explaining what constraints were the source of the denial.

If a constraint is not enumerated by a **LimitRange** it is not tracked.

kube-apiserver

The server is updated to be aware of **LimitRange** objects.

The constraints are only enforced if the kube-apiserver is started as follows:

```
$ kube-apiserver -admission_control=LimitRanger
```

kubectl

kubectl is modified to support the **LimitRange** resource.

`kubectl describe` provides a human-readable output of limits.

For example,

```
$ kubectl namespace myspace
$ kubectl create -f examples/limitrange/limit-range.json
$ kubectl get limits
NAME
limits
$ kubectl describe limits limits
Name:   limits
Type    Resource  Min Max
----    --------  --- ---
Pod   memory    1Mi 1Gi
Pod   cpu   250m  2
Container cpu   250m  2
Container memory    1Mi 1Gi
```

Future Enhancements: Define limits for a particular pod or container.

In the current proposal, the **LimitRangeItem** matches purely on **LimitRangeItem.Type**

It is expected we will want to define limits for particular pods or containers by name/uid and label/field selector.

To make a **LimitRangeItem** more restrictive, we will intend to add these additional restrictions at a future point in time.

Admission control plugin: ResourceQuota

Background

This document proposes a system for enforcing hard resource usage limits per namespace as part of admission control.

Model Changes

A new resource, **ResourceQuota**, is introduced to enumerate hard resource limits in a Kubernetes namespace.

A new resource, **ResourceQuotaUsage**, is introduced to support atomic updates of a **ResourceQuota** status.

```
// The following identify resource constants for Kubernetes object types
const (
  // Pods, number
  ResourcePods ResourceName = "pods"
  // Services, number
  ResourceServices ResourceName = "services"
  // ReplicationControllers, number
  ResourceReplicationControllers ResourceName = "replicationcontrollers"
  // ResourceQuotas, number
  ResourceQuotas ResourceName = "resourcequotas"
)

// ResourceQuotaSpec defines the desired hard limits to enforce for Quota
type ResourceQuotaSpec struct {
  // Hard is the set of desired hard limits for each named resource
  Hard ResourceList `json:"hard,omitempty"`
}

// ResourceQuotaStatus defines the enforced hard limits and observed use
type ResourceQuotaStatus struct {
  // Hard is the set of enforced hard limits for each named resource
  Hard ResourceList `json:"hard,omitempty"`
  // Used is the current observed total usage of the resource in the namespace
  Used ResourceList `json:"used,omitempty"`
}

// ResourceQuota sets aggregate quota restrictions enforced per namespace
type ResourceQuota struct {
  TypeMeta   `json:",inline"`
  ObjectMeta `json:"metadata,omitempty"`

  // Spec defines the desired quota
  Spec ResourceQuotaSpec `json:"spec,omitempty"`

  // Status defines the actual enforced quota and its current usage
  Status ResourceQuotaStatus `json:"status,omitempty"`
}

// ResourceQuotaUsage captures system observed quota status per namespace
// It is used to enforce atomic updates of a backing ResourceQuota.Status field in
type ResourceQuotaUsage struct {
  TypeMeta   `json:",inline"`
  ObjectMeta `json:"metadata,omitempty"`

  // Status defines the actual enforced quota and its current usage
  Status ResourceQuotaStatus `json:"status,omitempty"`
}

// ResourceQuotaList is a list of ResourceQuota items
type ResourceQuotaList struct {
  TypeMeta `json:",inline"`
```

```
  ListMeta `json:"metadata,omitempty"`

  // Items is a list of ResourceQuota objects
  Items []ResourceQuota `json:"items"`
}
```

AdmissionControl plugin: ResourceQuota

The **ResourceQuota** plug-in introspects all incoming admission requests.

It makes decisions by evaluating the incoming object against all defined **ResourceQuota.Status.Hard** resource limits in the request namespace. If acceptance of the resource would cause the total usage of a named resource to exceed its hard limit, the request is denied.

The following resource limits are imposed as part of core Kubernetes at the namespace level:

ResourceName	Description
cpu	Total cpu usage
memory	Total memory usage
pods	Total number of pods
services	Total number of services
replicationcontrollers	Total number of replication controllers
resourcequotas	Total number of resource quotas

Any resource that is not part of core Kubernetes must follow the resource naming convention prescribed by Kubernetes.

This means the resource must have a fully-qualified name (i.e. mycompany.org/shinynewresource)

If the incoming request does not cause the total usage to exceed any of the enumerated hard resource limits, the plug-in will post a **ResourceQuotaUsage** document to the server to atomically update the observed usage based on the previously read **ResourceQuota.ResourceVersion**. This keeps incremental usage atomically consistent, but does introduce a bottleneck (intentionally) into the system.

To optimize system performance, it is encouraged that all resource quotas are tracked on the same **ResourceQuota** document. As a result, its encouraged to actually impose a cap on the total number of individual quotas that are tracked in the **Namespace** to 1 by explicitly capping it in **ResourceQuota** document.

kube-apiserver

The server is updated to be aware of **ResourceQuota** objects.

The quota is only enforced if the kube-apiserver is started as follows:

```
$ kube-apiserver -admission_control=ResourceQuota
```

kube-controller-manager

A new controller is defined that runs a synch loop to calculate quota usage across the namespace.

ResourceQuota usage is only calculated if a namespace has a **ResourceQuota** object.

If the observed usage is different than the recorded usage, the controller sends a **ResourceQuotaUsage** resource to the server to atomically update.

The synchronization loop frequency will control how quickly DELETE actions are recorded in the system and usage is ticked down.

To optimize the synchronization loop, this controller will WATCH on Pod resources to track DELETE events, and in response, recalculate usage. This is because a Pod deletion will have the most impact on observed cpu and memory usage in the system, and we anticipate this being the resource most closely running at the prescribed quota limits.

kubectl

kubectl is modified to support the **ResourceQuota** resource.

`kubectl describe` provides a human-readable output of quota.

For example,

```
$ kubectl namespace myspace
$ kubectl create -f examples/resourcequota/resource-quota.json
$ kubectl get quota
NAME
myquota
$ kubectl describe quota myquota
Name: myquota
Resource  Used  Hard
--------  ----  ----
cpu 100m  20
memory  0 1.5Gb
pods  1 10
replicationControllers  1 10
services  2 3
```

This directory contains diagrams for the clustering design doc.

This depends on the `seqdiag` utility. Assuming you have a non-borked python install, this should be installable with

```
pip install seqdiag
```

Just call `make` to regenerate the diagrams.

Building with Docker

If you are on a Mac or your pip install is messed up, you can easily build with docker.

```
make docker
```

The first run will be slow but things should be fast after that.

To clean up the docker containers that are created (and other cruft that is left around) you can run `make docker-clean`.

If you are using boot2docker and get warnings about clock skew (or if things aren't building for some reason) then you can fix that up with `make fix-clock-skew`.

Automatically rebuild on file changes

If you have the fswatch utility installed, you can have it monitor the file system and automatically rebuild when files have changed. Just do a `make watch`.

Clustering in Kubernetes

Overview

The term "clustering" refers to the process of having all members of the kubernetes cluster find and trust each other. There are multiple different ways to achieve clustering with different security and usability profiles. This document attempts to lay out the user experiences for clustering that Kubernetes aims to address.

Once a cluster is established, the following is true:

1. **Master -> Node** The master needs to know which nodes can take work and what their current status is wrt capacity.
2. **Location** The master knows the name and location of all of the nodes in the cluster.
    - For the purposes of this doc, location and name should be enough information so that the master can open a TCP connection to the Node. Most probably we will make this either an IP address or a DNS name. It is going to be important to be consistent here (master must be able to reach kubelet on that DNS name) so that we can verify certificates appropriately.
3. **Target AuthN** A way to securely talk to the kubelet on that node. Currently we call out to the kubelet over HTTP. This should be over HTTPS and the master should know what CA to trust for that node.
4. **Caller AuthN/Z** This would be the master verifying itself (and permissions) when calling the node. Currently, this is only used to collect statistics as authorization isn't critical. This may change in the future though.
5. **Node -> Master** The nodes currently talk to the master to know which pods have been assigned to them and to publish events.
6. **Location** The nodes must know where the master is at.
7. **Target AuthN** Since the master is assigning work to the nodes, it is critical that they verify whom they are talking to.
8. **Caller AuthN/Z** The nodes publish events and so must be authenticated to the master. Ideally this authentication is specific to each node so that authorization can be narrowly scoped. The details of the work to run (including things like environment variables) might be considered sensitive and should be locked down also.

Note: While the description here refers to a singular Master, in the future we should enable multiple Masters operating in an HA mode. While the "Master" is currently the combination of the API Server, Scheduler and Controller Manager, we will restrict ourselves to thinking about the main API and policy engine -- the API Server.

Current Implementation

A central authority (generally the master) is responsible for determining the set of machines which are members of the cluster. Calls to create and remove worker nodes in the cluster are restricted to this single authority, and any other requests to add or remove worker nodes are rejected. (1.i).

Communication from the master to nodes is currently over HTTP and is not secured or authenticated in any way. (1.ii, 1.iii).

The location of the master is communicated out of band to the nodes. For GCE, this is done via Salt. Other cluster instructions/scripts use other methods. (2.i)

Currently most communication from the node to the master is over HTTP. When it is done over HTTPS there is currently no verification of the cert of the master (2.ii).

Currently, the node/kubelet is authenticated to the master via a token shared across all nodes. This token is distributed out of band (using Salt for GCE) and is optional. If it is not present then the kubelet is unable to publish events to the master. (2.iii)

Our current mix of out of band communication doesn't meet all of our needs from a security point of view and is difficult to set up and configure.

Proposed Solution

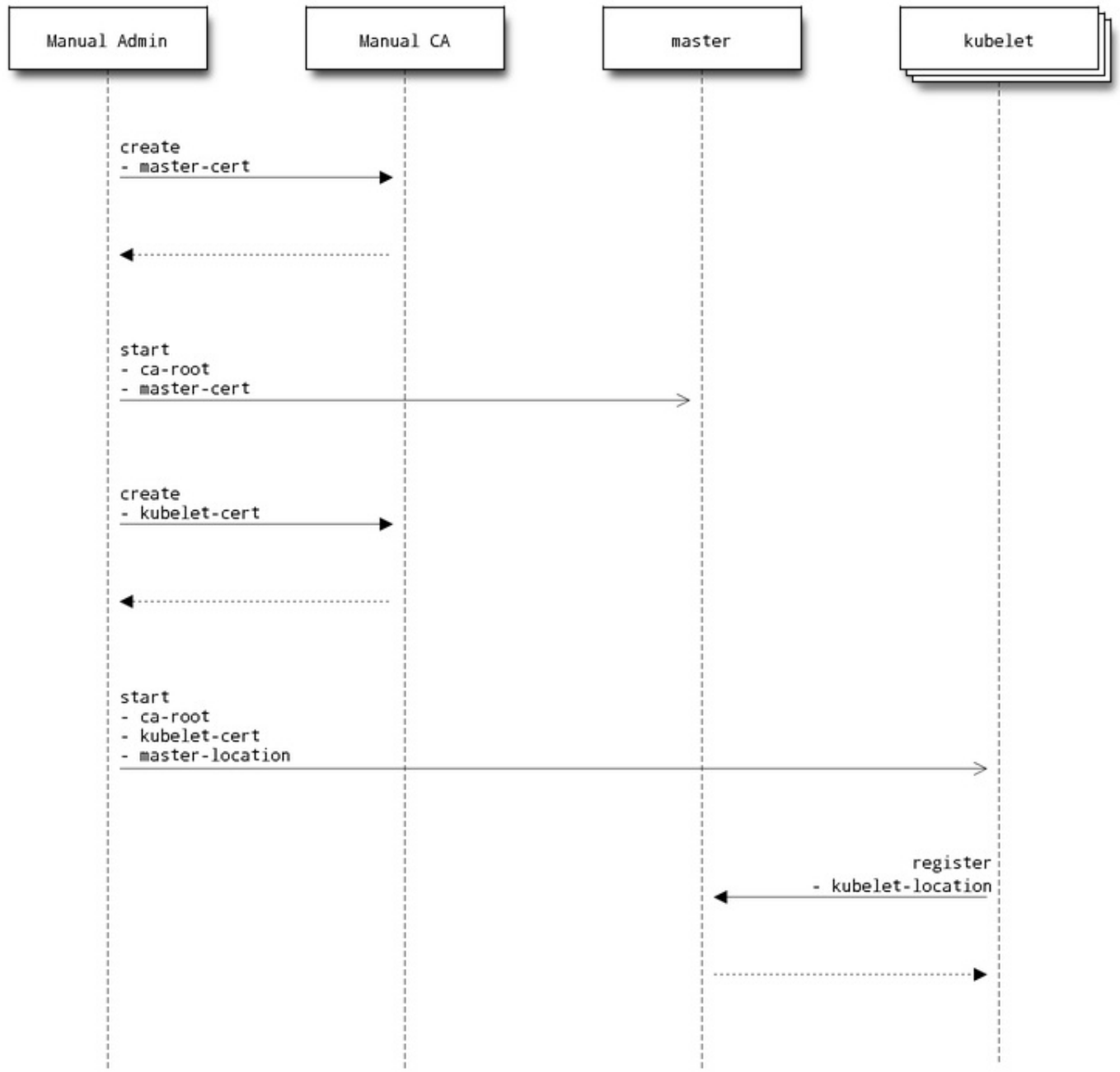
The proposed solution will provide a range of options for setting up and maintaining a secure Kubernetes cluster. We want to both allow for centrally controlled systems (leveraging pre-existing trust and configuration systems) or more ad-hoc automagic systems that are incredibly easy to set up.

The building blocks of an easier solution:

- **Move to TLS** We will move to using TLS for all intra-cluster communication. We will explicitly idenitfy the trust chain (the set of trusted CAs) as opposed to trusting the system CAs. We will also use client certificates for all AuthN.
- [optional] **API driven CA** Optionally, we will run a CA in the master that will mint certificates for the nodes/kubelets. There will be pluggable policies that will automatically approve certificate requests here as appropriate.
- **CA approval policy** This is a pluggable policy object that can automatically approve CA signing requests. Stock policies will include `always-reject`, `queue` and `insecure-always-approve`. With `queue` there would be an API for evaluating and accepting/rejecting requests. Cloud providers could implement a policy here that verifies other out of band information and automatically approves/rejects based on other external factors.
- **Scoped Kubelet Accounts** These accounts are per-minion and (optionally) give a minion permission to register itself.
    - To start with, we'd have the kubelets generate a cert/account in the form of `kubelet:<host>`. To start we would then hard code policy such that we give that particular account appropriate permissions. Over time, we can make the policy engine more generic.
- [optional] **Bootstrap API endpoint** This is a helper service hosted outside of the Kubernetes cluster that helps with initial discovery of the master.

Static Clustering

In this sequence diagram there is out of band admin entity that is creating all certificates and distributing them. It is also making sure that the kubelets know where to find the master. This provides for a lot of control but is more difficult to set up as lots of information must be communicated outside of Kubernetes.

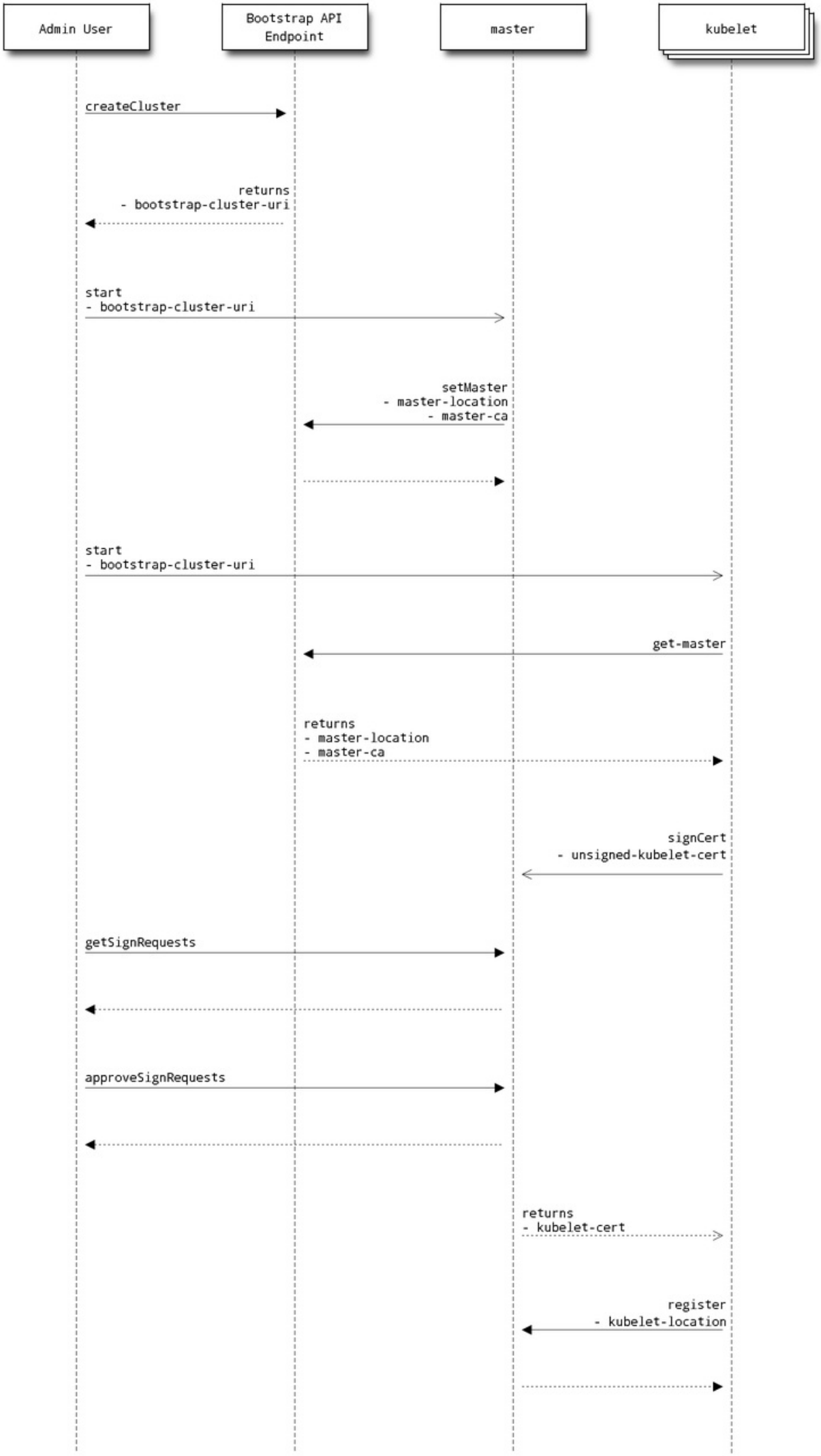


Static Sequence Diagram

Dynamic Clustering

This diagram dynamic clustering using the bootstrap API endpoint. That API endpoint is used to both find the location of the master and communicate the root CA for the master.

This flow has the admin manually approving the kubelet signing requests. This is the `queue` policy defined above.This manual intervention could be replaced by code that can verify the signing requests via other means.


Admin User
Bootstrap API Endpoint
master
kubelet
createCluster
returns
- bootstrap-cluster-uri
start
- bootstrap-cluster-uri
setMaster
- master-location
- master-ca
start
- bootstrap-cluster-uri
get-master
returns
- master-location
- master-ca
signCert
- unsigned-kubelet-cert
getSignRequests
approveSignRequests
returns
- kubelet-cert
register
- kubelet-location

Dynamic Sequence Diagram

Kubernetes Event Compression

This document captures the design of event compression.

Background

Kubernetes components can get into a state where they generate tons of events which are identical except for the timestamp. For example, when pulling a non-existing image, Kubelet will repeatedly generate `image_not_existing` and `container_is_waiting` events until upstream components correct the image. When this happens, the spam from the repeated events makes the entire event mechanism useless. It also appears to cause memory pressure in etcd (see #3853).

Proposal

Each binary that generates events (for example, `kubelet`) should keep track of previously generated events so that it can collapse recurring events into a single event instead of creating a new instance for each new event.

Event compression should be best effort (not guaranteed). Meaning, in the worst case, `n` identical (minus timestamp) events may still result in `n` event entries.

Design

Instead of a single Timestamp, each event object contains the following fields: * `FirstTimestamp util.Time` * The date/time of the first occurrence of the event. * `LastTimestamp util.Time` * The date/time of the most recent occurrence of the event. * On first occurrence, this is equal to the FirstTimestamp. * `Count int` * The number of occurrences of this event between FirstTimestamp and LastTimestamp * On first occurrence, this is 1.

Each binary that generates events: * Maintains a historical record of previously generated events: * Implmented with "Least Recently Used Cache" in `pkg/client/record/events_cache.go`. * The key in the cache is generated from the event object minus timestamps/count/transient fields, specifically the following events fields are used to construct a unique key for an event: * `event.Source.Component` * `event.Source.Host` * `event.InvolvedObject.Kind` * `event.InvolvedObject.Namespace` * `event.InvolvedObject.Name` * `event.InvolvedObject.UID` * `event.InvolvedObject.APIVersion` * `event.Reason` * `event.Message` * The LRU cache is capped at 4096 events. That means if a component (e.g. kubelet) runs for a long period of time and generates tons of unique events, the previously generated events cache will not grow unchecked in memory. Instead, after 4096 unique events are generated, the oldest events are evicted from the cache. * When an event is generated, the previously generated events cache is checked (see `pkg/client/record/event.go`). * If the key for the new event matches the key for a previously generated event (meaning all of the above fields match between the new event and some previously generated event), then the event is considered to be a duplicate and the existing event entry is updated in etcd: * The new PUT (update) event API is called to update the existing event entry in etcd with the new last seen timestamp and count. * The event is also updated in the previously generated events cache with an incremented count, updated last seen timestamp, name, and new resource version (all required to issue a future event update). * If the key for the new event does not match the key for any previously generated event (meaning none of the above fields match between the new event and any previously generated events), then the event is considered to be new/unique and a new event entry is created in etcd: * The usual POST/create event API is called to create a new event entry in etcd. * An entry for the event is also added to the previously generated events cache.

Issues/Risks

- Compression is not guaranteed, because each component keeps track of event history in memory
- An application restart causes event history to be cleared, meaning event history is not preserved across application restarts and compression will not occur across component restarts.
- Because an LRU cache is used to keep track of previously generated events, if too many unique events are generated, old events will be evicted from the cache, so events will only be compressed until they age out of the events cache, at which point any new instance of the event will cause a new entry to be created in etcd.

Example

Sample kubectl output

```
FIRSTSEEN                          LASTSEEN                           COUNT
Thu, 12 Feb 2015 01:13:02 +0000   Thu, 12 Feb 2015 01:13:02 +0000   1
Thu, 12 Feb 2015 01:13:09 +0000   Thu, 12 Feb 2015 01:13:09 +0000   1
Thu, 12 Feb 2015 01:13:09 +0000   Thu, 12 Feb 2015 01:13:09 +0000   1
Thu, 12 Feb 2015 01:13:09 +0000   Thu, 12 Feb 2015 01:13:09 +0000   1
Thu, 12 Feb 2015 01:13:05 +0000   Thu, 12 Feb 2015 01:13:12 +0000   4
Thu, 12 Feb 2015 01:13:05 +0000   Thu, 12 Feb 2015 01:13:12 +0000   4
Thu, 12 Feb 2015 01:13:05 +0000   Thu, 12 Feb 2015 01:13:12 +0000   4
Thu, 12 Feb 2015 01:13:05 +0000   Thu, 12 Feb 2015 01:13:12 +0000   4
Thu, 12 Feb 2015 01:13:05 +0000   Thu, 12 Feb 2015 01:13:12 +0000   4
Thu, 12 Feb 2015 01:13:20 +0000   Thu, 12 Feb 2015 01:13:20 +0000   1
Thu, 12 Feb 2015 01:13:20 +0000   Thu, 12 Feb 2015 01:13:20 +0000   1
```

This demonstrates what would have been 20 separate entries (indicating scheduling failure) collapsed/compressed down to 5 entries.

Related Pull Requests/Issues

- Issue #4073: Compress duplicate events
- PR #4157: Add "Update Event" to Kubernetes API
- PR #4206: Modify Event struct to allow compressing multiple recurring events in to a single event
- PR #4306: Compress recurring events in to a single event to optimize etcd storage
- PR #4444: Switch events history to use LRU cache instead of map

Identifiers and Names in Kubernetes

A summarization of the goals and recommendations for identifiers in Kubernetes. Described in GitHub issue #199.

Definitions

UID : A non-empty, opaque, system-generated value guaranteed to be unique in time and space; intended to distinguish between historical occurrences of similar entities.

Name : A non-empty string guaranteed to be unique within a given scope at a particular time; used in resource URLs; provided by clients at creation time and encouraged to be human friendly; intended to facilitate creation idempotence and space-uniqueness of singleton objects, distinguish distinct entities, and reference particular entities across operations.

rfc1035/rfc1123 label (DNS_LABEL) : An alphanumeric (a-z, and 0-9) string, with a maximum length of 63 characters, with the '-' character allowed anywhere except the first or last character, suitable for use as a hostname or segment in a domain name

rfc1035/rfc1123 subdomain (DNS_SUBDOMAIN) : One or more lowercase rfc1035/rfc1123 labels separated by '.' with a maximum length of 253 characters

rfc4122 universally unique identifier (UUID) : A 128 bit generated value that is extremely unlikely to collide across time and space and requires no central coordination

Objectives for names and UIDs

1. Uniquely identify (via a UID) an object across space and time
2. Uniquely name (via a name) an object across space
3. Provide human-friendly names in API operations and/or configuration files
4. Allow idempotent creation of API resources (#148) and enforcement of space-uniqueness of singleton objects
5. Allow DNS names to be automatically generated for some objects

General design

1. When an object is created via an API, a Name string (a DNS_SUBDOMAIN) must be specified. Name must be non-empty and unique within the apiserver. This enables idempotent and space-unique creation operations. Parts of the system (e.g. replication controller) may join strings (e.g. a base name and a random suffix) to create a unique Name. For situations where generating a name is impractical, some or all objects may support a param to auto-generate a name. Generating random names will defeat idempotency.

- Examples: "guestbook.user", "backend-x4eb1"

2. When an object is created via an api, a Namespace string (a DNS_SUBDOMAIN? format TBD via #1114) may be specified. Depending on the API receiver, namespaces might be validated (e.g. apiserver might ensure that the namespace actually exists). If a namespace is not specified, one will be assigned by the API receiver. This assignment policy might vary across API receivers (e.g. apiserver might have a default, kubelet might generate something semi-random).

- Example: "api.k8s.example.com"

3. Upon acceptance of an object via an API, the object is assigned a UID (a UUID). UID must be non-empty and unique across space and time.

- Example: "01234567-89ab-cdef-0123-456789abcdef"

Case study: Scheduling a pod

Pods can be placed onto a particular node in a number of ways. This case study demonstrates how the above design can be applied to satisfy the objectives.

A pod scheduled by a user through the apiserver

1. A user submits a pod with Namespace="" and Name="guestbook" to the apiserver.

2. The apiserver validates the input.
3. A default Namespace is assigned.
4. The pod name must be space-unique within the Namespace.

5. Each container within the pod has a name which must be space-unique within the pod.

6. The pod is accepted.

7. A new UID is assigned.

8. The pod is bound to a node.

9. The kubelet on the node is passed the pod's UID, Namespace, and Name.

10. Kubelet validates the input.

11. Kubelet runs the pod.
12. Each container is started up with enough metadata to distinguish the pod from whence it came.
13. Each attempt to run a container is assigned a UID (a string) that is unique across time.
    - This may correspond to Docker's container ID.

A pod placed by a config file on the node

1. A config file is stored on the node, containing a pod with UID="", Namespace="", and Name="cadvisor".

2. Kubelet validates the input.
3. Since UID is not provided, kubelet generates one.
4. Since Namespace is not provided, kubelet generates one.
    1. The generated namespace should be deterministic and cluster-unique for the source, such as a hash of the hostname and file path.
        - E.g. Namespace="file-f4231812554558a718a01ca942782d81"
5. Kubelet runs the pod.
6. Each container is started up with enough metadata to distinguish the pod from whence it came.
7. Each attempt to run a container is assigned a UID (a string) that is unique across time.
    1. This may correspond to Docker's container ID.

Design: Limit direct access to etcd from within Kubernetes

All nodes have effective access of "root" on the entire Kubernetes cluster today because they have access to etcd, the central data store. The kubelet, the service proxy, and the nodes themselves have a connection to etcd that can be used to read or write any data in the system. In a cluster with many hosts, any container or user that gains the ability to write to the network device that can reach etcd, on any host, also gains that access.

- The Kubelet and Kube Proxy currently rely on an efficient "wait for changes over HTTP" interface get their current state and avoid missing changes
- This interface is implemented by etcd as the "watch" operation on a given key containing useful data

Options:

1. Do nothing
2. Introduce an HTTP proxy that limits the ability of nodes to access etcd
    1. Prevent writes of data from the kubelet
    2. Prevent reading data not associated with the client responsibilities
    3. Introduce a security token granting access
3. Introduce an API on the apiserver that returns the data a node Kubelet and Kube Proxy needs
    1. Remove the ability of nodes to access etcd via network configuration
    2. Provide an alternate implementation for the event writing code Kubelet
    3. Implement efficient "watch for changes over HTTP" to offer comparable function with etcd
    4. Ensure that the apiserver can scale at or above the capacity of the etcd system.
    5. Implement authorization scoping for the nodes that limits the data they can view
4. Implement granular access control in etcd
    1. Authenticate HTTP clients with client certificates, tokens, or BASIC auth and authorize them for read only access
    2. Allow read access of certain subpaths based on what the requestor's tokens are

Evaluation:

Option 1 would be considered unacceptable for deployment in a multi-tenant or security conscious environment. It would be acceptable in a low security deployment where all software is trusted. It would be acceptable in proof of concept environments on a single machine.

Option 2 would require implementing an http proxy that for 2-1 could block POST/PUT/DELETE requests (and potentially HTTP method tunneling parameters accepted by etcd). 2-2 would be more complicated and would require filtering operations based on deep understanding of the etcd API *and* the underlying schema. It would be possible, but involve extra software.

Option 3 would involve extending the existing apiserver to return pods associated with a given node over an HTTP "watch for changes" mechanism, which is already implemented. Proper security would involve checking that the caller is authorized to access that data - one imagines a per node token, key, or SSL certificate that could be used to authenticate and then authorize access to only the data belonging to that node. The current event publishing mechanism from the kubelet would also need to be replaced with a secure API endpoint or a change to a polling model. The apiserver would also need to be able to function in a horizontally scalable mode by changing or fixing the "operations" queue to work in a stateless, scalable model. In practice, the amount of traffic even a large Kubernetes deployment would drive towards an apiserver would be tens of requests per second (500 hosts, 1 request per host every minute) which is negligible if well implemented. Implementing this would also decouple the data store schema from the nodes, allowing a different data store technology to be added in the future without affecting existing nodes. This would also expose that data to other consumers for their own purposes (monitoring, implementing service discovery).

Option 4 would involve extending etcd to support access control. Administrators would need to authorize nodes to connect to etcd, and expose network routability directly to etcd. The mechanism for handling this authentication and authorization would be different than the authorization used by Kubernetes controllers and API clients. It would not be possible to completely replace etcd as a data store without also implementing a new Kubelet config endpoint.

Preferred solution:

Implement the first parts of option 3 - an efficient watch API for the pod, service, and endpoints data for the Kubelet and Kube Proxy. Authorization and authentication are planned in the future - when a solution is available, implement a custom authorization scope that allows API access to be restricted to only the data about a single node or the service endpoint data.

In general, option 4 is desirable in addition to option 3 as a mechanism to further secure the store to infrastructure components that must access it.

Caveats

In all four options, compromise of a host will allow an attacker to imitate that host. For attack vectors that are reproducible from inside containers (privilege escalation), an attacker can distribute himself to other hosts by requesting new containers be spun up. In scenario 1, the cluster is totally compromised immediately. In 2-1, the attacker can view all information about the cluster including keys or authorization data defined with pods. In 2-2 and 3, the attacker must still distribute himself in order to get access to a large subset of information, and cannot see other data that is potentially located in etcd like side storage or system configuration. For attack vectors that are not exploits, but instead allow network access to etcd, an attacker in 2ii has no ability to spread his influence, and is instead restricted to the subset of information on the host. For 3-5, they can do nothing they could not do already (request access to the nodes / services endpoint) because the token is not visible to them on the host.

Labels

Labels are key/value pairs identifying client/user-defined attributes (and non-primitive system-generated attributes) of API objects, which are stored and returned as part of the metadata of those objects. Labels can be used to organize and to select subsets of objects according to these attributes.

Each object can have a set of key/value labels set on it, with at most one label with a particular key.

```
"labels": {
  "key1" : "value1",
  "key2" : "value2"
}
```

Unlike names and UIDs, labels do not provide uniqueness. In general, we expect many objects to carry the same label(s).

Via a *label selector*, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.

Label selectors may also be used to associate policies with sets of objects.

We also plan to make labels available inside pods and lifecycle hooks.

Valid label keys are comprised of two segments - prefix and name - separated by a slash (`/`). The name segment is required and must be a DNS label: 63 characters or less, all lowercase, beginning and ending with an alphanumeric character (`[a-z0-9]`), with dashes (`-`) and alphanumerics between. The prefix and slash are optional. If specified, the prefix must be a DNS subdomain (a series of DNS labels separated by dots (`.`), not longer than 253 characters in total.

If the prefix is omitted, the label key is presumed to be private to the user. System components which use labels must specify a prefix. The `kubernetes.io` prefix is reserved for kubernetes core components.

Motivation

Service deployments and batch processing pipelines are often multi-dimensional entities (e.g., multiple partitions or deployments, multiple release tracks, multiple tiers, multiple micro-services per tier). Management often requires cross-cutting operations, which breaks encapsulation of strictly hierarchical representations, especially rigid hierarchies determined by the infrastructure rather than by users. Labels enable users to map their own organizational structures onto system objects in a loosely coupled fashion, without requiring clients to store these mappings.

Label selectors

Label selectors permit very simple filtering by label keys and values. The simplicity of label selectors is deliberate. It is intended to facilitate transparency for humans, easy set overlap detection, efficient indexing, and reverse-indexing (i.e., finding all label selectors matching an object's labels - https://github.com/GoogleCloudPlatform/kubernetes/issues/1348).

Currently the system supports selection by exact match of a map of keys and values. Matching objects must have all of the specified labels (both keys and values), though they may have additional labels as well.

We are in the process of extending the label selection specification (see selector.go and https://github.com/GoogleCloudPlatform/kubernetes/issues/341) to support conjunctions of requirements of the following forms:

```
key1 in (value11, value12, ...)
key1 not in (value11, value12, ...)
key1 exists
```

LIST and WATCH operations may specify label selectors to filter the sets of objects returned using a query parameter: `?labels=key1%3Dvalue1,key2%3Dvalue2,...`. We may extend such filtering to DELETE operations in the future.

Kubernetes also currently supports two objects that use label selectors to keep track of their members, `services` and `replicationControllers`: - `service`: A service is a configuration unit for the proxies that run on every worker node. It is named and points to one or more pods. - `replicationController`: A replication controller ensures that a specified number of pod "replicas" are running at any one time. If there are too many, it'll kill some. If there are too few, it'll start more.

The set of pods that a `service` targets is defined with a label selector. Similarly, the population of pods that a `replicationController` is monitoring is also defined with a label selector.

For management convenience and consistency, `services` and `replicationControllers` may themselves have labels and would generally carry the labels their corresponding pods have in common.

In the future, label selectors will be used to identify other types of distributed service workers, such as worker pool members or peers in a distributed application.

Individual labels are used to specify identifying metadata, and to convey the semantic purposes/roles of pods of containers. Examples of typical pod label keys include `service`, `environment` (e.g., with values `dev`, `qa`, or `production`), `tier` (e.g., with values `frontend` or `backend`), and `track` (e.g., with values `daily` or `weekly`), but you are free to develop your own conventions.

Sets identified by labels and label selectors could be overlapping (think Venn diagrams). For instance, a service might target all pods with `tier in (frontend), environment in (prod)`. Now say you have 10 replicated pods that make up this tier. But you want to be able to 'canary' a new version of this component. You could set up a `replicationController` (with `replicas` set to 9) for the bulk of the replicas with labels `tier=frontend, environment=prod, track=stable` and another `replicationController` (with `replicas` set to 1) for the canary with labels `tier=frontend, environment=prod, track=canary`. Now the service is covering both the canary and non-canary pods. But you can mess with the `replicationControllers` separately to test things out, monitor the results, etc.

Note that the superset described in the previous example is also heterogeneous. In long-lived, highly available, horizontally scaled, distributed, continuously evolving service applications, heterogeneity is inevitable, due to canaries, incremental rollouts, live reconfiguration, simultaneous updates and auto-scaling, hardware upgrades, and so on.

Pods (and other objects) may belong to multiple sets simultaneously, which enables representation of service substructure and/or superstructure. In particular, labels are intended to facilitate the creation of non-hierarchical, multi-dimensional deployment structures. They are useful for a variety of management purposes

(e.g., configuration, deployment) and for application introspection and analysis (e.g., logging, monitoring, alerting, analytics). Without the ability to form sets by intersecting labels, many implicitly related, overlapping flat sets would need to be created, for each subset and/or superset desired, which would lose semantic information and be difficult to keep consistent. Purely hierarchically nested sets wouldn't readily support slicing sets across different dimensions.

Pods may be removed from these sets by changing their labels. This flexibility may be used to remove pods from service for debugging, data recovery, etc.

Since labels can be set at pod creation time, no separate set add/remove operations are necessary, which makes them easier to use than manual set management. Additionally, since labels are directly attached to pods and label selectors are fairly simple, it's easy for users and for clients and tools to determine what sets they belong to (i.e., they are reversible). OTOH, with sets formed by just explicitly enumerating members, one would (conceptually) need to search all sets to determine which ones a pod belonged to.

Labels vs. annotations

We'll eventually index and reverse-index labels for efficient queries and watches, use them to sort and group in UIs and CLIs, etc. We don't want to pollute labels with non-identifying, especially large and/or structured, data. Non-identifying information should be recorded using annotations.

Networking

Model and motivation

Kubernetes deviates from the default Docker networking model. The goal is for each pod to have an IP in a flat shared networking namespace that has full communication with other physical computers and containers across the network. IP-per-pod creates a clean, backward-compatible model where pods can be treated much like VMs or physical hosts from the perspectives of port allocation, networking, naming, service discovery, load balancing, application configuration, and migration.

OTOH, dynamic port allocation requires supporting both static ports (e.g., for externally accessible services) and dynamically allocated ports, requires partitioning centrally allocated and locally acquired dynamic ports, complicates scheduling (since ports are a scarce resource), is inconvenient for users, complicates application configuration, is plagued by port conflicts and reuse and exhaustion, requires non-standard approaches to naming (e.g., etcd rather than DNS), requires proxies and/or redirection for programs using standard naming/addressing mechanisms (e.g., web browsers), requires watching and cache invalidation for address/port changes for instances in addition to watching group membership changes, and obstructs container/pod migration (e.g., using CRIU). NAT introduces additional complexity by fragmenting the addressing space, which breaks self-registration mechanisms, among other problems.

With the IP-per-pod model, all user containers within a pod behave as if they are on the same host with regard to networking. They can all reach each other’s ports on localhost. Ports which are published to the host interface are done so in the normal Docker way. All containers in all pods can talk to all other containers in all other pods by their 10-dot addresses.

In addition to avoiding the aforementioned problems with dynamic port allocation, this approach reduces friction for applications moving from the world of uncontainerized apps on physical or virtual hosts to containers within pods. People running application stacks together on the same host have already figured out how to make ports not conflict (e.g., by configuring them through environment variables) and have arranged for clients to find them.

The approach does reduce isolation between containers within a pod -- ports could conflict, and there couldn't be private ports across containers within a pod, but applications requiring their own port spaces could just run as separate pods and processes requiring private communication could run within the same container. Besides, the premise of pods is that containers within a pod share some resources (volumes, cpu, ram, etc.) and therefore expect and tolerate reduced isolation. Additionally, the user can control what containers belong to the same pod whereas, in general, they don't control what pods land together on a host.

When any container calls SIOCGIFADDR, it sees the IP that any peer container would see them coming from -- each pod has its own IP address that other pods can know. By making IP addresses and ports the same within and outside the containers and pods, we create a NAT-less, flat address space. "ip addr show" should work as expected. This would enable all existing naming/discovery mechanisms to work out of the box, including self-registration mechanisms and applications that distribute IP addresses. (We should test that with etcd and perhaps one other option, such as Eureka (used by Acme Air) or Consul.) We should be optimizing for inter-pod network communication. Within a pod, containers are more likely to use communication through volumes (e.g., tmpfs) or IPC.

This is different from the standard Docker model. In that mode, each container gets an IP in the 172-dot space and would only see that 172-dot address from SIOCGIFADDR. If these containers connect to another container the peer would see the connect coming from a different IP than the container itself knows. In short - you can never self-register anything from a container, because a container can not be reached on its private IP.

An alternative we considered was an additional layer of addressing: pod-centric IP per container. Each container would have its own local IP address, visible only within that pod. This would perhaps make it easier for containerized applications to move from physical/virtual hosts to pods, but would be more complex to implement (e.g., requiring a bridge per pod, split-horizon/VP DNS) and to reason about, due to the additional layer of address translation, and would break self-registration and IP distribution mechanisms.

Current implementation

For the Google Compute Engine cluster configuration scripts, advanced routing is set up so that each VM has an extra 256 IP addresses that get routed to it. This is in addition to the 'main' IP address assigned to the VM that is NAT-ed for Internet access. The networking bridge (called `cbr0` to differentiate it from `docker0`) is set up outside of Docker proper and only does NAT for egress network traffic that isn't aimed at the virtual network.

Ports mapped in from the 'main IP' (and hence the internet if the right firewall rules are set up) are proxied in user mode by Docker. In the future, this should be done with `iptables` by either the Kubelet or Docker: Issue #15.

We start Docker with: DOCKER_OPTS="--bridge cbr0 --iptables=false"

We set up this bridge on each node with SaltStack, in container_bridge.py.

```
cbr0:
  container_bridge.ensure:
  - cidr: {{ grains['cbr-cidr'] }}
...
grains:
  roles:
  - kubernetes-pool
  cbr-cidr: $MINION_IP_RANGE
```

We make these addresses routable in GCE:

```
gcloud compute routes add "${MINION_NAMES[$i]}" \
  --project "${PROJECT}" \
  --destination-range "${MINION_IP_RANGES[$i]}" \
  --network "${NETWORK}" \
  --next-hop-instance "${MINION_NAMES[$i]}" \
  --next-hop-instance-zone "${ZONE}" &
```

The minion IP ranges are /24s in the 10-dot space.

GCE itself does not know anything about these IPs, though.

These are not externally routable, though, so containers that need to communicate with the outside world need to use host networking. To set up an external IP that forwards to the VM, it will only forward to the VM's primary IP (which is assigned to no pod). So we use docker's -p flag to map published ports to the main interface. This has the side effect of disallowing two pods from exposing the same port. (More discussion on this in Issue #390.)

We create a container to use for the pod network namespace -- a single loopback device and a single veth device. All the user's containers get their network namespaces from this pod networking container.

Docker allocates IP addresses from a bridge we create on each node, using its “container” networking mode.

1. Create a normal (in the networking sense) container which uses a minimal image and runs a command that blocks forever. This is not a user-defined container, and gets a special well-known name.

- creates a new network namespace (netns) and loopback device
- creates a new pair of veth devices and binds them to the netns
- auto-assigns an IP from docker’s IP range

2. Create the user containers and specify the name of the pod infra container as their “POD” argument. Docker finds the PID of the command running in the pod infra container and attaches to the netns and ipcns of that PID.

Other networking implementation examples

With the primary aim of providing IP-per-pod-model, other implementations exist to serve the purpose outside of GCE. - OpenVSwitch with GRE/VxLAN - Flannel

Challenges and future work

Docker API

Right now, docker inspect doesn't show the networking configuration of the containers, since they derive it from another container. That information should be exposed somehow.

External IP assignment

We want to be able to assign IP addresses externally from Docker (Docker issue #6743) so that we don't need to statically allocate fixed-size IP ranges to each node, so that IP addresses can be made stable across pod infra container restarts (Docker issue #2801), and to facilitate pod migration. Right now, if the pod infra container dies, all the user containers must be stopped and restarted because the netns of the pod infra container will change on restart, and any subsequent user container restart will join that new netns, thereby not being able to see its peers. Additionally, a change in IP address would encounter DNS caching/TTL problems. External IP assignment would also simplify DNS support (see below).

Naming, discovery, and load balancing

In addition to enabling self-registration with 3rd-party discovery mechanisms, we'd like to setup DDNS automatically (Issue #146). hostname, $HOSTNAME, etc. should return a name for the pod (Issue #298), and gethostbyname should be able to resolve names of other pods. Probably we need to set up a DNS resolver to do the latter (Docker issue #2267), so that we don't need to keep /etc/hosts files up to date dynamically.

Service endpoints are currently found through environment variables. Both Docker-links-compatible variables and kubernetes-specific variables ({NAME}_SERVICE_HOST and {NAME}_SERVICE_BAR) are supported, and resolve to ports opened by the service proxy. We don't actually use the Docker ambassador pattern to link containers because we don't require applications to identify all clients at configuration time, yet. While services today are managed by the service proxy, this is an implementation detail that applications should not rely on. Clients should instead use the service portal IP (which the above environment variables will resolve to). However, a flat service namespace doesn't scale and environment variables don't permit dynamic updates, which complicates service deployment by imposing implicit ordering constraints. We intend to register each service portal IP in DNS, and for that to become the preferred resolution protocol.

We'd also like to accommodate other load-balancing solutions (e.g., HAProxy), non-load-balanced services (Issue #260), and other types of groups (worker pools, etc.). Providing the ability to Watch a label selector applied to pod addresses would enable efficient monitoring of group membership, which could be directly consumed or synced with a discovery mechanism. Event hooks (Issue #140) for join/leave events would probably make this even easier.

External routability

We want traffic between containers to use the pod IP addresses across nodes. Say we have Node A with a container IP space of 10.244.1.0/24 and Node B with a container IP space of 10.244.2.0/24. And we have Container A1 at 10.244.1.1 and Container B1 at 10.244.2.1. We want Container A1 to talk to Container B1 directly with no NAT. B1 should see the "source" in the IP packets of 10.244.1.1 -- not the "primary" host IP for Node A. That means that we want to turn off NAT for traffic between containers (and also between VMs and containers).

We'd also like to make pods directly routable from the external internet. However, we can't yet support the extra container IPs that we've provisioned talking to the internet directly. So, we don't map external IPs to the container IPs. Instead, we solve that problem by having traffic that isn't to the internal network (! 10.0.0.0/8) get NATed through the primary host IP address so that it can get 1:1 NATed by the GCE networking when talking to the internet. Similarly, incoming traffic from the internet has to get NATed/proxied through the host IP.

So we end up with 3 cases:

1. Container -> Container or Container <-> VM. These should use 10. addresses directly and there

should be no NAT.

2. Container -> Internet. These have to get mapped to the primary host IP so that GCE knows how to egress that traffic. There is actually 2 layers of NAT here: Container IP -> Internal Host IP -> External Host IP. The first level happens in the guest with IP tables and the second happens as part of GCE networking. The first one (Container IP -> internal host IP) does dynamic port allocation while the second maps ports 1:1.

3. Internet -> Container. This also has to go through the primary host IP and also has 2 levels of NAT, ideally. However, the path currently is a proxy with (External Host IP -> Internal Host IP -> Docker) -> (Docker -> Container IP). Once issue #15 is closed, it should be External Host IP -> Internal Host IP -> Container IP. But to get that second arrow we have to set up the port forwarding iptables rules per mapped port.

Another approach could be to create a new host interface alias for each pod, if we had a way to route an external IP to it. This would eliminate the scheduling constraints resulting from using the host's IP address.

IPv6

IPv6 would be a nice option, also, but we can't depend on it yet. Docker support is in progress: Docker issue #2974, Docker issue #6923, Docker issue #6975. Additionally, direct ipv6 assignment to instances doesn't appear to be supported by major cloud providers (e.g., AWS EC2, GCE) yet. We'd happily take pull requests from people running Kubernetes on bare metal, though. :-)

Security in Kubernetes

Kubernetes should define a reasonable set of security best practices that allows processes to be isolated from each other, from the cluster infrastructure, and which preserves important boundaries between those who manage the cluster, and those who use the cluster.

While Kubernetes today is not primarily a multi-tenant system, the long term evolution of Kubernetes will increasingly rely on proper boundaries between users and administrators. The code running on the cluster must be appropriately isolated and secured to prevent malicious parties from affecting the entire cluster.

High Level Goals

1. Ensure a clear isolation between the container and the underlying host it runs on
2. Limit the ability of the container to negatively impact the infrastructure or other containers
3. Principle of Least Privilege - ensure components are only authorized to perform the actions they need, and limit the scope of a compromise by limiting the capabilities of individual components
4. Reduce the number of systems that have to be hardened and secured by defining clear boundaries between components
5. Allow users of the system to be cleanly separated from administrators
6. Allow administrative functions to be delegated to users where necessary
7. Allow applications to be run on the cluster that have "secret" data (keys, certs, passwords) which is properly abstracted from "public" data.

Use cases

Roles:

We define "user" as a unique identity accessing the Kubernetes API server, which may be a human or an automated process. Human users fall into the following categories:

1. k8s admin - administers a kubernetes cluster and has access to the undelying components of the system
2. k8s project administrator - administrates the security of a small subset of the cluster
3. k8s developer - launches pods on a kubernetes cluster and consumes cluster resources

Automated process users fall into the following categories:

1. k8s container user - a user that processes running inside a container (on the cluster) can use to access other cluster resources indepedent of the human users attached to a project
2. k8s infrastructure user - the user that kubernetes infrastructure components use to perform cluster functions with clearly defined roles

Description of roles:

- Developers:
- write pod specs.
- making some of their own images, and using some "community" docker images
- know which pods need to talk to which other pods
- decide which pods should be share files with other pods, and which should not.
- reason about application level security, such as containing the effects of a local-file-read exploit in a webserver pod.
- do not often reason about operating system or organizational security.

- are not necessarily comfortable reasoning about the security properties of a system at the level of detail of Linux Capabilities, SELinux, AppArmor, etc.

- Project Admins:
- allocate identity and roles within a namespace
- reason about organizational security within a namespace
    - don't give a developer permissions that are not needed for role.
    - protect files on shared storage from unnecessary cross-team access

- are less focused about application security

- Administrators:
- are less focused on application security. Focused on operating system security.
- protect the node from bad actors in containers, and properly-configured innocent containers from bad actors in other containers.
- comfortable reasoning about the security properties of a system at the level of detail of Linux Capabilities, SELinux, AppArmor, etc.
- decides who can use which Linux Capabilities, run privileged containers, use hostDir, etc.
    - e.g. a team that manages Ceph or a mysql server might be trusted to have raw access to storage devices in some organizations, but teams that develop the applications at higher layers would not.

Proposed Design

A pod runs in a *security context* under a *service account* that is defined by an administrator or project administrator, and the *secrets* a pod has access to is limited by that *service account*.

1. The API should authenticate and authorize user actions authn and authz
2. All infrastructure components (kubelets, kube-proxies, controllers, scheduler) should have an infrastructure user that they can authenticate with and be authorized to perform only the functions they require against the API.
3. Most infrastructure components should use the API as a way of exchanging data and changing the system, and only the API should have access to the underlying data store (etcd)
4. When containers run on the cluster and need to talk to other containers or the API server, they should be identified and authorized clearly as an autonomous process via a service account
5. If the user who started a long-lived process is removed from access to the cluster, the process should be able to continue without interruption
6. If the user who started processes are removed from the cluster, administrators may wish to terminate their processes in bulk
7. When containers run with a service account, the user that created / triggered the service account behavior must be associated to the container's action
8. When container processes runs on the cluster, they should run in a security context that isolates those processes via Linux user security, user namespaces, and permissions.
9. Administrators should be able to configure the cluster to automatically confine all container processes as a non-root, randomly assigned UID
10. Administrators should be able to ensure that container processes within the same namespace are all assigned the same unix user UID
11. Administrators should be able to limit which developers and project administrators have access to higher privilege actions
12. Project administrators should be able to run pods within a namespace under different security contexts, and developers must be able to specify which of the available security contexts they may use
13. Developers should be able to run their own images or images from the community and expect those images to run correctly
14. Developers may need to ensure their images work within higher security requirements specified by administrators
15. When available, Linux kernel user namespaces can be used to ensure 5.2 and 5.4 are met.
16. When application developers want to share filesytem data via distributed filesystems, the Unix user ids on those filesystems must be consistent across different container processes
17. Developers should be able to define secrets that are automatically added to the containers when pods are run
18. Secrets are files injected into the container whose values should not be displayed within a pod. Examples:
    1. An SSH private key for git cloning remote data
    2. A client certificate for accessing a remote system
    3. A private key and certificate for a web server
    4. A .kubeconfig file with embedded cert / token data for accessing the Kubernetes master
    5. A .dockercfg file for pulling images from a protected registry
19. Developers should be able to define the pod spec so that a secret lands in a specific location
20. Project administrators should be able to limit developers within a namespace from viewing or modify secrets (anyone who can launch an arbitrary pod can view secrets)
21. Secrets are generally not copied from one namespace to another when a developer's application definitions are copied

Related design discussion

- Authorization and authentication https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/design/access.html
- Secret distribution via files https://github.com/GoogleCloudPlatform/kubernetes/pull/2030
- Docker secrets https://github.com/docker/docker/pull/6697
- Docker vault https://github.com/docker/docker/issues/10310

Specific Design Points

TODO: authorization, authentication

Isolate the data store from the minions and supporting infrastructure

Access to the central data store (etcd) in Kubernetes allows an attacker to run arbitrary containers on hosts, to gain access to any protected information stored in either volumes or in pods (such as access tokens or shared secrets provided as environment variables), to intercept and redirect traffic from running services by inserting middlemen, or to simply delete the entire history of the custer.

As a general principle, access to the central data store should be restricted to the components that need full control over the system and which can apply appropriate authorization and authentication of change requests. In the future, etcd may offer granular access control, but that granularity will require an administrator to understand the schema of the data to properly apply security. An administrator must be able to properly secure Kubernetes at a policy level, rather than at an implementation level, and schema changes over time should not risk unintended security leaks.

Both the Kubelet and Kube Proxy need information related to their specific roles - for the Kubelet, the set of pods it should be running, and for the Proxy, the set of services and endpoints to load balance. The Kubelet also needs to provide information about running pods and historical termination data. The access pattern for both Kubelet and Proxy to load their configuration is an efficient "wait for changes" request over HTTP. It should be possible to limit the Kubelet and Proxy to only access the information they need to perform their roles and no more.

The controller manager for Replication Controllers and other future controllers act on behalf of a user via delegation to perform automated maintenance on Kubernetes resources. Their ability to access or modify resource state should be strictly limited to their intended duties and they should be prevented from accessing information not pertinent to their role. For example, a replication controller needs only to create a copy of a known pod configuration, to determine the running state of an existing pod, or to delete an existing pod that it created - it does not need to know the contents or current state of a pod, nor have access to any data in the pods attached volumes.

The Kubernetes pod scheduler is responsible for reading data from the pod to fit it onto a minion in the cluster. At a minimum, it needs access to view the ID of a pod (to craft the binding), its current state, any resource information necessary to identify placement, and other data relevant to concerns like anti-affinity, zone or region preference, or custom logic. It does not need the ability to modify pods or see other resources, only to create bindings. It should not need the ability to delete bindings unless the scheduler takes control of relocating components on failed hosts (which could be implemented by a separate component that can delete bindings but not create them). The scheduler may need read access to user or project-container information to determine preferential location (underspecified at this time).

Security Contexts

Abstract

A security context is a set of constraints that are applied to a container in order to achieve the following goals (from security design):

1. Ensure a clear isolation between container and the underlying host it runs on
2. Limit the ability of the container to negatively impact the infrastructure or other containers

Background

The problem of securing containers in Kubernetes has come up before and the potential problems with container security are well known. Although it is not possible to completely isolate Docker containers from their hosts, new features like user namespaces make it possible to greatly reduce the attack surface.

Motivation

Container isolation

In order to improve container isolation from host and other containers running on the host, containers should only be granted the access they need to perform their work. To this end it should be possible to take advantage of Docker features such as the ability to add or remove capabilities and assign MCS labels to the container process.

Support for user namespaces has recently been merged into Docker's libcontainer project and should soon surface in Docker itself. It will make it possible to assign a range of unprivileged uids and gids from the host to each container, improving the isolation between host and container and between containers.

External integration with shared storage

In order to support external integration with shared storage, processes running in a Kubernetes cluster should be able to be uniquely identified by their Unix UID, such that a chain of ownership can be established. Processes in pods will need to have consistent UID/GID/SELinux category labels in order to access shared disks.

Constraints and Assumptions

- It is out of the scope of this document to prescribe a specific set of constraints to isolate containers from their host. Different use cases need different settings.
- The concept of a security context should not be tied to a particular security mechanism or platform (ie. SELinux, AppArmor)
- Applying a different security context to a scope (namespace or pod) requires a solution such as the one proposed for service accounts.

Use Cases

In order of increasing complexity, following are example use cases that would be addressed with security contexts:

1. Kubernetes is used to run a single cloud application. In order to protect nodes from containers:
    - All containers run as a single non-root user
    - Privileged containers are disabled
    - All containers run with a particular MCS label
    - Kernel capabilities like CHOWN and MKNOD are removed from containers
2. Just like case #1, except that I have more than one application running on the Kubernetes cluster.
    - Each application is run in its own namespace to avoid name collisions
    - For each application a different uid and MCS label is used
3. Kubernetes is used as the base for a PAAS with multiple projects, each project represented by a namespace.
    - Each namespace is associated with a range of uids/gids on the node that are mapped to uids/gids on containers using linux user namespaces.
    - Certain pods in each namespace have special privileges to perform system actions such as talking back to the server for deployment, run docker builds, etc.
    - External NFS storage is assigned to each namespace and permissions set using the range of uids/gids assigned to that namespace.

Proposed Design

Overview

A *security context* consists of a set of constraints that determine how a container is secured before getting created and run. It has a 1:1 correspondence to a service account. A *security context provider* is passed to the Kubelet so it can have a chance to mutate Docker API calls in order to apply the security context.

It is recommended that this design be implemented in two phases:

1. Implement the security context provider extension point in the Kubelet so that a default security context can be applied on container run and creation.
2. Implement a security context structure that is part of a service account. The default context provider can then be used to apply a security context based on the service account associated with the pod.

Security Context Provider

The Kubelet will have an interface that points to a `SecurityContextProvider`. The `SecurityContextProvider` is invoked before creating and running a given container:

```
type SecurityContextProvider interface {
    // ModifyContainerConfig is called before the Docker createContainer call.
    // The security context provider can make changes to the Config with which
    // the container is created.
    // An error is returned if it's not possible to secure the container as
    // requested with a security context.
    ModifyContainerConfig(pod *api.BoundPod, container *api.Container, config *doc

    // ModifyHostConfig is called before the Docker runContainer call.
    // The security context provider can make changes to the HostConfig, affecting
    // security options, whether the container is privileged, volume binds, etc.
    // An error is returned if it's not possible to secure the container as reques
    // with a security context.
    ModifyHostConfig(pod *api.BoundPod, container *api.Container, hostConfig *dock
}
```

If the value of the SecurityContextProvider field on the Kubelet is nil, the kubelet will create and run the container as it does today.

Security Context

A security context has a 1:1 correspondence to a service account and it can be included as part of the service account resource. Following is an example of an initial implementation:

```
// SecurityContext specifies the security constraints associated with a service ac
type SecurityContext struct {
    // user is the uid to use when running the container
    User int

    // AllowPrivileged indicates whether this context allows privileged mode conta
    AllowPrivileged bool

    // AllowedVolumeTypes lists the types of volumes that a container can bind
    AllowedVolumeTypes []string

    // AddCapabilities is the list of Linux kernel capabilities to add
    AddCapabilities []string
```

```
    // RemoveCapabilities is the list of Linux kernel capabilities to remove
    RemoveCapabilities []string

    // Isolation specifies the type of isolation required for containers
    // in this security context
    Isolation ContainerIsolationSpec
}

// ContainerIsolationSpec indicates intent for container isolation
type ContainerIsolationSpec struct {
    // Type is the container isolation type (None, Private)
    Type ContainerIsolationType

    // FUTURE: IDMapping specifies how users and groups from the host will be mapp
    IDMapping *IDMapping
}

// ContainerIsolationType is the type of container isolation for a security contex
type ContainerIsolationType string

const (
    // ContainerIsolationNone means that no additional consraints are added to
    // containers to isolate them from their host
    ContainerIsolationNone ContainerIsolationType = "None"

    // ContainerIsolationPrivate means that containers are isolated in process
    // and storage from their host and other containers.
    ContainerIsolationPrivate ContainerIsolationType = "Private"
)

// IDMapping specifies the requested user and group mappings for containers
// associated with a specific security context
type IDMapping struct {
    // SharedUsers is the set of user ranges that must be unique to the entire clu
    SharedUsers []IDMappingRange

    // SharedGroups is the set of group ranges that must be unique to the entire c
    SharedGroups []IDMappingRange

    // PrivateUsers are mapped to users on the host node, but are not necessarily
    // unique to the entire cluster
    PrivateUsers []IDMappingRange

    // PrivateGroups are mapped to groups on the host node, but are not necessaril
    // unique to the entire cluster
    PrivateGroups []IDMappingRange
}

// IDMappingRange specifies a mapping between container IDs and node IDs
type IDMappingRange struct {
    // ContainerID is the starting container ID
    ContainerID int

    // HostID is the starting host ID
    HostID int

    // Length is the length of the ID range
    Length int
}
```

Security Context Lifecycle

The lifecycle of a security context will be tied to that of a service account. It is expected that a service account with a default security context will be created for every Kubernetes namespace (without administrator intervention). If resources need to be allocated when creating a security context (for example, assign a range of host uids/gids), a pattern such as finalizers can be used before declaring the security context / service account / namespace ready for use.

Reaching the API

Ports and IPs Served On

The Kubernetes API is served by the Kubernetes APIServer process. Typically, there is one of these running on a single kubernetes-master node.

By default the Kubernetes APIserver serves HTTP on 3 ports: 1. Localhost Port - serves HTTP - default is port 8080, change with `-port` flag. - defaults IP is localhost, change with `-address` flag. - no authentication or authorization checks in HTTP - protected by need to have host access 2. ReadOnly Port - default is port 7080, change with `-read_only_port` - default IP is first non-localhost network interface, change with `-public_address_override` - serves HTTP - no authentication checks in HTTP - only GET requests are allowed. - requests are rate limited 3. Secure Port - default is port 6443, change with `-secure_port` - default IP is first non-localhost network interface, change with `-public_address_override` - serves HTTPS. Set cert with `-tls_cert_file` and key with `-tls_private_key_file`. - uses token-file based authentication. - uses policy-based authorization.

Proxies and Firewall rules

Additionally, in typical configurations (i.e. GCE), there is a proxy (nginx) running on the same machine as the apiserver process. The proxy serves HTTPS protected by Basic Auth on port 443, and proxies to the apiserver on localhost:8080. Typically, firewall rules will allow HTTPS access to port 443.

The above are defaults and reflect how Kubernetes is deployed to GCE using kube-up.sh. Other cloud providers may vary.

Use Cases vs IP:Ports

There are three differently configured serving ports because there are a variety of uses cases: 1. Clients outside of a Kubernetes cluster, such as human running `kubectl` on desktop machine. Currently, accesses the Localhost Port via a proxy (nginx) running on the `kubernetes-master` machine. Proxy uses Basic Auth. 2. Processes running in Containers on Kubernetes that need to do read from the apiserver. Currently, these can use Readonly Port. 3. Scheduler and Controller-manager processes, which need to do read-write API operations. Currently, these have to run on the operations on the apiserver. Currently, these have to run on the same host as the apiserver and use the Localhost Port. 4. Kubelets, which need to do read-write API operations and are necessarily on different machines than the apiserver. Kubelet uses the Secure Port to get their pods, to find the services that a pod can see, and to write events. Credentials are distributed to kubelets at cluster setup time.

Expected changes

- Policy will limit the actions kubelets can do via the authed port.
- Kube-proxy currently uses the readonly port to read services and endpoints, but will eventually use the auth port.
- Kubelets may change from token-based authentication to cert-based-auth.
- Scheduler and Controller-manager will use the Secure Port too. They will then be able to run on different machines than the apiserver.
- A general mechanism will be provided for giving credentials to pods.
- The Readonly Port will no longer be needed and will be removed.
- Clients, like kubectl, will all support token-based auth, and the Localhost will no longer be needed, and will not be the default. However, the localhost port may continue to be an option for installations that want to do their own auth proxy.

Annotations

We have labels for identifying metadata.

It is also useful to be able to attach arbitrary non-identifying metadata, for retrieval by API clients such as tools, libraries, etc. This information may be large, may be structured or unstructured, may include characters not permitted by labels, etc. Such information would not be used for object selection and therefore doesn't belong in labels.

Like labels, annotations are key-value maps.

```
"annotations": {
  "key1" : "value1",
  "key2" : "value2"
}
```

Possible information that could be recorded in annotations: * fields managed by a declarative configuration layer, to distinguish them from client- and/or server-set default values and other auto-generated fields, fields set by auto-sizing/auto-scaling systems, etc., in order to facilitate merging * build/release/image information (timestamps, release ids, git branch, PR numbers, image hashes, registry address, etc.) * pointers to logging/monitoring/analytics/audit repos * client library/tool information (e.g. for debugging purposes -- name, version, build info) * other user and/or tool/system provenance info, such as URLs of related objects from other ecosystem components * lightweight rollout tool metadata (config and/or checkpoints) * phone/pager number(s) of person(s) responsible, or directory entry where that info could be found, such as a team website

Yes, this information could be stored in an external database or directory, but that would make it much harder to produce shared client libraries and tools for deployment, management, introspection, etc.

API Conventions

The conventions of the Kubernetes API (and related APIs in the ecosystem) are intended to ease client development and ensure that configuration mechanisms can be implemented that work across a diverse set of use cases consistently.

The general style of the Kubernetes API is RESTful - clients create, update, delete, or retrieve a description of an object via the standard HTTP verbs (POST, PUT, DELETE, and GET) - and those APIs preferentially accept and return JSON. Kubernetes also exposes additional endpoints for non-standard verbs and allows alternative content types.

The following terms are defined:

- **Endpoint** a URL on an HTTP server that modifies, retrieves, or transforms a Resource.
- **Resource** an object manipulated via an HTTP action in an API
- **Kind** a resource has a string that identifies the schema of the JSON used (e.g. a "Car" and a "Dog" would have different attributes and properties)

Types of Resources

All API resources are either:

1. **Objects** represents a physical or virtual construct in the system. An API object is a record of intent - once created, the system will work to ensure that resource exists. All API objects have common metadata intended for client use.
2. **Lists** are collections of **objects** of one or more types. Lists have a limited set of common metadata. All lists use the "items" field to contain the array of objects they return. Each resource kind should have an endpoint that returns the full set of resources, as well as zero or more endpoints that return subsets of the full list.

In addition, all lists that return objects with labels should support label filtering (see labels.html, and most lists should support filtering by fields.

TODO: Describe field filtering below or in a separate doc.

The standard REST verbs (defined below) MUST return singular JSON objects. Some API endpoints may deviate from the strict REST pattern and return resources that are not singular JSON objects, such as streams of JSON objects or unstructured text log data.

Resources

All singular JSON resources returned by an API MUST have the following fields:

- kind: a string that identifies the schema this object should have
- apiVersion: a string that identifies the version of the schema the object should have

Objects

Metadata

Every object MUST have the following metadata in a nested object field called "metadata":

- namespace: a namespace is a DNS compatible subdomain that objects are subdivided into. The default namespace is 'default'. See namespaces.html for more.
- name: a string that uniquely identifies this object within the current namespace (see identifiers.html). This value is used in the path when retrieving an individual object.
- uid: a unique in time and space value (typically an RFC 4122 generated identifier, see identifiers.html) used to distinguish between objects with the same name that have been deleted and recreated

Every object SHOULD have the following metadata in a nested object field called "metadata":

- resourceVersion: a string that identifies the internal version of this object that can be used by clients to determine when objects have changed. This value MUST be treated as opaque by clients and passed unmodified back to the server. Clients should not assume that the resource version has meaning across namespaces, different kinds of resources, or different servers. (see concurrency control, below, for more details)
- creationTimestamp: a string representing an RFC 3339 date of the date and time an object was created
- labels: a map of string keys and values that can be used to organize and categorize objects (see labels.html)
- annotations: a map of string keys and values that can be used by external tooling to store and retrieve arbitrary metadata about this object (see annotations.html)

Labels are intended for organizational purposes by end users (select the pods that match this label query). Annotations enable third party automation and tooling to decorate objects with additional metadata for their own use.

Spec and Status

By convention, the Kubernetes API makes a distinction between the specification of the desired state of a resource (a nested object field called "spec") and the status of the resource at the current time (a nested object field called "status"). The specification is persisted in stable storage with the API object and reflects user input. The status is generated at runtime and summarizes the current effect that the spec has on the system.

For example, a pod object has a "spec" field that defines how the pod should be run. The pod also has a "status" field that shows details about what is happening on the host that is running the containers in the pod (if available) and a summarized "status" string that can guide callers as to the overall state of their pod.

When a new version of an object is POSTed or PUT, the "spec" is updated and available immediately. Over time the system will work to bring the "status" into line with the "spec". The system will drive toward the most recent "spec" regardless of previous versions of that stanza. In other words, if a value is changed from 2 to 5 in one PUT and then back down to 3 in another PUT the system is not required to 'touch base' at 5 before changing the "status" to 3.

The PUT and POST verbs will ignore the "status" values. Otherwise, PUT expects the whole object to be specified. Therefore, if a field is omitted it is assumed that the client wants to clear that field's value.

Modification of just part of an object may be achieved by GETting the resource, modifying part of the spec, labels, or annotations, and then PUTting it back. See concurrency control, below, regarding read-modify-write consistency when using this pattern.

Lists of named subobjects preferred over maps

Discussed in #2004 and elsewhere. There are no maps of subobjects in any API objects. Instead, the convention is to use a list of subobjects containing name fields.

For example:

```yaml
ports:
  - name: www
    containerPort: 80
```

vs.

```yaml
ports:
  www:
    containerPort: 80
```

This rule maintains the invariant that all JSON/YAML keys are fields in API objects. The only exceptions are pure maps in the API (currently, labels, selectors, and annotations), as opposed to sets of subobjects.

Lists

Every list SHOULD have the following metadata in a nested object field called "metadata":

- resourceVersion: a string that identifies the common version of the objects returned by in a list. This value MUST be treated as opaque by clients and passed unmodified back to the server. A resource version is only valid within a single namespace on a single kind of resource.

Special Resources

Kubernetes MAY return two resources from any API endpoint in special circumstances. Clients SHOULD handle these types of objects when appropriate.

A "Status" object SHOULD be returned by an API when an operation is not successful (when the server would return a non 2xx HTTP status code). The status object contains fields for humans and machine consumers of the API to determine failures. The information in the status object supplements, but does not override, the HTTP status code's meaning.

TODO: More details (refer to another doc for details)

Synthetic Resources

An API may represent a single object in different ways for different clients, or transform an object after certain transitions in the system occur. In these cases, one request object may have two representations available as different resource kinds. An example is a Pod, which represents the intent of the user to run a container with certain parameters. When Kubernetes schedules the Pod, it creates a Binding object that ties that Pod to a single host in the system. After this occurs, the pod is represented by two distinct resources - under the original Pod resource the user created, as well as in a BoundPods object that the host may query but not update.

Verbs on Resources

API resources should use the traditional REST pattern:

- GET /<resourceNamePlural> - Retrieve a list of type <resourceName>, e.g. GET /pods returns a list of Pods.
- POST /<resourceNamePlural> - Create a new resource from the JSON object provided by the client.
- GET /<resourceNamePlural>/<name> - Retrieves a single resource with the given name, e.g. GET /pods/first returns a Pod named 'first'.
- DELETE /<resourceNamePlural>/<name> - Delete the single resource with the given name.
- PUT /<resourceNamePlural>/<name> - Update or create the resource with the given name with the JSON object provided by the client.

Kubernetes by convention exposes additional verbs as new endpoints with singular names. Examples:

- GET /watch/<resourceNamePlural> - Receive a stream of JSON objects corresponding to changes made to any resource of the given kind over time.
- GET /watch/<resourceNamePlural>/<name> - Receive a stream of JSON objects corresponding to changes made to the named resource of the given kind over time.
- GET /redirect/<resourceNamePlural>/<name> - If the named resource can be described by a URL, return an HTTP redirect to that URL instead of a JSON response. For example, a service exposes a port and IP address and a client could invoke the redirect verb to receive an HTTP 307 redirection to that port and IP.
- GET /proxy/<resourceNamePlural>/<name>/{subpath:*} - Proxy GET request to the named resource of the given kind. Can be used for example, to access log files on pods.

Support of additional verbs is not required for all object types.

TODO: more documentation of Watch

Idempotency

All compatible Kubernetes APIs MUST support "name idempotency" and respond with an HTTP status code 409 when a request is made to POST an object that has the same name as an existing object in the system. See identifiers.html for details.

TODO: name generation

Concurrency Control and Consistency

Read-modify-write consistency is accomplished with optimistic currency.

All resources have "resourceVersion" as part of their metadata. resourceVersion is a string that identifies the internal version of an object that can be used by clients to determine when objects have changed. It is changed by the server every time an object is modified. If resourceVersion is included with the PUT operation the system will verify that there have not been other successful mutations to the resource during a read/modify/write cycle, by verifying that the current value of resourceVersion matches the specified value.

The only way for a client to know the expected value of resourceVersion is to have received it from the server in response to a prior operation, typically a GET. This value MUST be treated as opaque by clients and passed unmodified back to the server. Clients should not assume that the resource version has meaning across namespaces, different kinds of resources, or different servers. Currently, the value of resourceVersion is set to match etcd's sequencer. You could think of it as a logical clock the API server can use to order requests. However, we expect the implementation of resourceVersion to change in the future, such as in the case we shard the state by kind and/or namespace, or port to another storage system.

APIs SHOULD set resourceVersion on retrieved resources, and support PUT idempotency by rejecting HTTP requests with a StatusConflict (409) HTTP status code where an HTTP header `If-Match: resourceVersion=` or `?resourceVersion=` query parameter are set and do not match the currently stored version of the resource. (Currently, the API simply uses the value from the PUT request body.) The correct client action at this point is to GET the resource again, apply the changes afresh and try submitting again.

This mechanism can be used to prevent races like the following:

```
Client #1                                   Client #2
GET Foo                                     GET Foo
Set Foo.Bar = "one"                         Set Foo.Baz = "two"
PUT Foo                                     PUT Foo
```

When these sequences occur in parallel, either the change to Foo.Bar or the change to Foo.Baz can be lost.

On the other hand, when specifying the resourceVersion, one of the PUTs will fail, since whichever write succeeds changes the resourceVersion for Foo.

resourceVersion may be used as a precondition for other operations (e.g., GET, DELETE) in the future, such as for read-after-write consistency in the presence of caching.

"Watch" operations specify resourceVersion using a query parameter. It is used to specify the point at which to begin watching the specified resources. This may be used to ensure that no mutations are missed between a GET of a resource (or list of resources) and a subsequent Watch, even if the current version of the resource is more recent. This is currently the main reason that list operations (GET on a collection) return resourceVersion.

TODO: better syntax?

Serialization Format

APIs may return alternative representations of any resource in response to an Accept header or under alternative endpoints, but the default serialization for input and output of API responses MUST be JSON.

All dates should be serialized as RFC3339 strings.

Selecting Fields

Some APIs may need to identify which field in a JSON object is invalid, or to reference a value to extract from a separate resource. The current recommendation is to use standard JavaScript syntax for accessing that field, assuming the JSON object was transformed into a JavaScript object.

Examples:

- Find the field "current" in the object "state" in the second item in the array "fields": `fields[0].state.current`

TODO: Plugins, extensions, nested kinds, headers

Status codes

The following status codes may be returned by the API.

TODO: Document when each of these codes is returned

Success codes

- `StatusOK`
- `StatusCreated`
- `StatusAccepted`
- `StatusNoContent`

Error codes

- `StatusNotFound`
- `StatusMethodNotAllowed`
- `StatusUnsupportedMediaType`
- `StatusNotAcceptable`
- `StatusBadRequest`
- `StatusUnauthorized`
- `StatusForbidden`
- `StatusRequestTimeout`
- `StatusConflict`
- `StatusPreconditionFailed`
- `StatusUnprocessableEntity`
- `StatusInternalServerError`
- `StatusServiceUnavailable`

TODO: also document API status strings, reasons, and causes

Events

TODO: Document events (refer to another doc for details)

API Documentation

API documentation can be found at http://kubernetes.io/third_party/swagger-ui/.

Namespaces

Namespaces help different projects, teams, or customers to share a kubernetes cluster. First, they provide a scope for Names. Second, as our access control code develops, it is expected that it will be convenient to attach authorization and other policy to namespaces.

Use of multiple namespaces is optional. For small teams, they may not be needed.

Namespaces are still under development. For now, the best documentation is the Namespaces Design Document.

Container with Kubernetes

Capabilities

By default, Docker containers are "unprivileged" and cannot, for example, run a Docker daemon inside a Docker container. We can have fine grain control over the capabilities using cap-add and cap-drop.More details here.

The relationship between Docker's capabilities and Linux capabilities

Docker's capabilities	Linux capabilities
SETPCAP	CAP_SETPCAP
SYS_MODULE	CAP_SYS_MODULE
SYS_RAWIO	CAP_SYS_RAWIO
SYS_PACCT	CAP_SYS_PACCT
SYS_ADMIN	CAP_SYS_ADMIN
SYS_NICE	CAP_SYS_NICE
SYS_RESOURCE	CAP_SYS_RESOURCE
SYS_TIME	CAP_SYS_TIME
SYS_TTY_CONFIG	CAP_SYS_TTY_CONFIG
MKNOD	CAP_MKNOD
AUDIT_WRITE	CAP_AUDIT_WRITE
AUDIT_CONTROL	CAP_AUDIT_CONTROL
MAC_OVERRIDE	CAP_MAC_OVERRIDE
MAC_ADMIN	CAP_MAC_ADMIN
NET_ADMIN	CAP_NET_ADMIN
SYSLOG	CAP_SYSLOG
CHOWN	CAP_CHOWN
NET_RAW	CAP_NET_RAW
DAC_OVERRIDE	CAP_DAC_OVERRIDE
FOWNER	CAP_FOWNER
DAC_READ_SEARCH	CAP_DAC_READ_SEARCH
FSETID	CAP_FSETID
KILL	CAP_KILL
SETGID	CAP_SETGID
SETUID	CAP_SETUID
LINUX_IMMUTABLE	CAP_LINUX_IMMUTABLE
NET_BIND_SERVICE	CAP_NET_BIND_SERVICE
NET_BROADCAST	CAP_NET_BROADCAST
IPC_LOCK	CAP_IPC_LOCK
IPC_OWNER	CAP_IPC_OWNER
SYS_CHROOT	CAP_SYS_CHROOT
SYS_PTRACE	CAP_SYS_PTRACE
SYS_BOOT	CAP_SYS_BOOT
LEASE	CAP_LEASE
SETFCAP	CAP_SETFCAP
WAKE_ALARM	CAP_WAKE_ALARM

BLOCK_SUSPEND CAP_BLOCK_SUSPEND

On Collaborative Development

Kubernetes is open source, but many of the people working on it do so as their day job. In order to avoid forcing people to be "at work" effectively 24/7, we want to establish some semi-formal protocols around development. Hopefully these rules make things go more smoothly. If you find that this is not the case, please complain loudly.

Patches welcome

First and foremost: as a potential contributor, your changes and ideas are welcome at any hour of the day or night, weekdays, weekends, and holidays. Please do not ever hesitate to ask a question or send a PR.

Timezones and calendars

For the time being, most of the people working on this project are in the US and on Pacific time. Any times mentioned henceforth will refer to this timezone. Any references to "work days" will refer to the US calendar.

Code reviews

All changes must be code reviewed. For non-maintainers this is obvious, since you can't commit anyway. But even for maintainers, we want all changes to get at least one review, preferably from someone who knows the areas the change touches. For non-trivial changes we may want two reviewers. The primary reviewer will make this decision and nominate a second reviewer, if needed. Except for trivial changes, PRs should sit for at least 2 hours to allow for wider review.

Most PRs will find reviewers organically. If a maintainer intends to be the primary reviewer of a PR they should set themselves as the assignee on GitHub and say so in a reply to the PR. Only the primary reviewer of a change should actually do the merge, except in rare cases (e.g. they are unavailable in a reasonable timeframe).

If a PR has gone 2 work days without an owner emerging, please poke the PR thread and ask for a reviewer to be assigned.

Except for rare cases, such as trivial changes (e.g. typos, comments) or emergencies (e.g. broken builds), maintainers should not merge their own changes.

Expect reviewers to request that you avoid common go style mistakes in your PRs.

Assigned reviews

Maintainers can assign reviews to other maintainers, when appropriate. The assignee becomes the shepherd for that PR and is responsible for merging the PR once they are satisfied with it or else closing it. The assignee might request reviews from non-maintainers.

Merge hours

Maintainers will do merges between the hours of 7:00 am Monday and 7:00 pm (19:00h) Friday. PRs that arrive over the weekend or on holidays will only be merged if there is a very good reason for it and if the code review requirements have been met.

There may be discussion an even approvals granted outside of the above hours, but merges will generally be deferred.

Holds

Any maintainer or core contributor who wants to review a PR but does not have time immediately may put a hold on a PR simply by saying so on the PR discussion and offering an ETA measured in single-digit days at most. Any PR that has a hold shall not be merged until the person who requested the hold acks the review, withdraws their hold, or is overruled by a preponderance of maintainers.

Development Guide

Releases and Official Builds

Official releases are built in Docker containers. Details are here. You can do simple builds and development with just a local Docker installation. If want to build go locally outside of docker, please continue below.

Go development environment

Kubernetes is written in Go programming language. If you haven't set up Go development environment, please follow this instruction to install go tool and set up GOPATH. Ensure your version of Go is at least 1.3.

Put kubernetes into GOPATH

We highly recommend to put kubernetes' code into your GOPATH. For example, the following commands will download kubernetes' code under the current user's GOPATH (Assuming there's only one directory in GOPATH.):

```
$ echo $GOPATH
/home/user/goproj
$ mkdir -p $GOPATH/src/github.com/GoogleCloudPlatform/
$ cd $GOPATH/src/github.com/GoogleCloudPlatform/
$ git clone https://github.com/GoogleCloudPlatform/kubernetes.git
```

The commands above will not work if there are more than one directory in `$GOPATH`.

(Obviously, clone your own fork of Kubernetes if you plan to do development.)

godep and dependency management

Kubernetes uses godep to manage dependencies. It is not strictly required for building Kubernetes but it is required when managing dependencies under the Godeps/ tree, and is required by a number of the build and test scripts. Please make sure that `godep` is installed and in your `$PATH`.

Installing godep

There are many ways to build and host go binaries. Here is an easy way to get utilities like `godep` installed:

1. Ensure that mercurial is installed on your system. (some of godep's dependencies use the mercurial source control system). Use `apt-get install mercurial` or `yum install mercurial` on Linux, or brew.sh on OS X, or download directly from mercurial.

2. Create a new GOPATH for your tools and install godep:

   ```
   export GOPATH=$HOME/go-tools
   mkdir -p $GOPATH
   go get github.com/tools/godep
   ```

3. Add $GOPATH/bin to your path. Typically you'd add this to your ~/.profile: ``` export GOPATH=$HOME/go-tools export PATH=*PATH* : GOPATH/bin ```

Using godep

Here is a quick summary of `godep`. `godep` helps manage third party dependencies by copying known versions into Godeps/_workspace. Here is the recommended way to set up your system. There are other ways that may work, but this is the easiest one I know of.

1. Devote a directory to this endeavor:

   ```
   export KPATH=$HOME/code/kubernetes
   mkdir -p $KPATH/src/github.com/GoogleCloudPlatform/kubernetes
   cd $KPATH/src/github.com/GoogleCloudPlatform/kubernetes
   git clone https://path/to/your/fork .
   # Or copy your existing local repo here. IMPORTANT: making a symlink doesn't
   ```

2. Set up your GOPATH.

   ```
   # Option A: this will let your builds see packages that exist elsewhere on y
   export GOPATH=$KPATH:$GOPATH
   # Option B: This will *not* let your local builds see packages that exist el
   export GOPATH=$KPATH
   # Option B is recommended if you're going to mess with the dependencies.
   ```

3. Populate your new $GOPATH.

   ```
   cd $KPATH/src/github.com/GoogleCloudPlatform/kubernetes
   godep restore
   ```

4. Next, you can either add a new dependency or update an existing one. ``` # To add a new dependency, run: cd $KPATH/src/github.com/GoogleCloudPlatform/kubernetes go get path/to/dependency godep save ./...

To update an existing dependency, do

cd $KPATH/src/github.com/GoogleCloudPlatform/kubernetes go get -u path/to/dependency godep update path/to/dependency

```
5) Before sending your PR, it's a good idea to sanity check that your Godeps.json

I (lavalamp) have sometimes found it expedient to manually fix the /Godeps/godeps.

Please send dependency updates in separate commits within your PR, for easier rev

## Hooks

Before committing any changes, please link/copy these hooks into your .git
directory. This will keep you from accidentally committing non-gofmt'd go code.
```

cd kubernetes/.git/hooks/ ln -s ../../hooks/prepare-commit-msg . ln -s ../../hooks/commit-msg .

```
## Unit tests
```

cd kubernetes hack/test-go.sh

```
Alternatively, you could also run:
```

cd kubernetes godep go test ./...

```
If you only want to run unit tests in one package, you could run ``godep go test``
```

$ cd kubernetes # step into kubernetes' directory. $ cd pkg/kubelet $ godep go test # some output from unit tests PASS ok github.com/GoogleCloudPlatform/kubernetes/pkg/kubelet 0.317s

```
## Coverage
```

cd kubernetes godep go tool cover -html=target/c.out

```
## Integration tests

You need an [etcd](https://github.com/coreos/etcd/releases/tag/v2.0.0) in your pat
```

cd kubernetes hack/test-integration.sh

```
## End-to-End tests

You can run an end-to-end test which will bring up a master and two minions, perfo
```

cd kubernetes hack/e2e-test.sh

```
Pressing control-C should result in an orderly shutdown but if something goes wron
```

go run hack/e2e.go --down ```

Flag options

See the flag definitions in `hack/e2e.go` for more options, such as reusing an existing cluster, here is an overview:

```
# Build binaries for testing
go run hack/e2e.go --build

# Create a fresh cluster.  Deletes a cluster first, if it exists
go run hack/e2e.go --up

# Create a fresh cluster at a specific release version.
go run hack/e2e.go --up --version=0.7.0

# Test if a cluster is up.
go run hack/e2e.go --isup

# Push code to an existing cluster
go run hack/e2e.go --push

# Push to an existing cluster, or bring up a cluster if it's down.
go run hack/e2e.go --pushup

# Run all tests
go run hack/e2e.go --test

# Run tests matching a glob.
go run hack/e2e.go --tests=...
```

Combining flags

```
# Flags can be combined, and their actions will take place in this order:
# -build, -push|-up|-pushup, -test|-tests=..., -down
# e.g.:
go run e2e.go -build -pushup -test -down

# -v (verbose) can be added if you want streaming output instead of only
# seeing the output of failed commands.

# -ctl can be used to quickly call kubectl against your e2e cluster. Useful for
# cleaning up after a failed test or viewing logs.
go run e2e.go -ctl='get events'
go run e2e.go -ctl='delete pod foobar'
```

Testing out flaky tests

Instructions here

Add/Update dependencies

Kubernetes uses godep to manage dependencies. To add or update a package, please follow the instructions on godep's document.

To add a new package `foo/bar`:

- Make sure the kubernetes' root directory is in $GOPATH/github.com/GoogleCloudPlatform/kubernetes
- Run `godep restore` to make sure you have all dependancies pulled.
- Download foo/bar into the first directory in GOPATH: `go get foo/bar`.
- Change code in kubernetes to use `foo/bar`.
- Run `godep save ./...` under kubernetes' root directory.

To update a package `foo/bar`:

- Make sure the kubernetes' root directory is in $GOPATH/github.com/GoogleCloudPlatform/kubernetes
- Run `godep restore` to make sure you have all dependancies pulled.
- Update the package with `go get -u foo/bar`.
- Change code in kubernetes accordingly if necessary.
- Run `godep update foo/bar` under kubernetes' root directory.

Keeping your development fork in sync

One time after cloning your forked repo:

```
git remote add upstream https://github.com/GoogleCloudPlatform/kubernetes.git
```

Then each time you want to sync to upstream:

```
git fetch upstream
git rebase upstream/master
```

Regenerating the CLI documentation

```
hack/run-gendocs.sh
```

Hunting flaky tests in Kubernetes

Sometimes unit tests are flaky. This means that due to (usually) race conditions, they will occasionally fail, even though most of the time they pass.

We have a goal of 99.9% flake free tests. This means that there is only one flake in one thousand runs of a test.

Running a test 1000 times on your own machine can be tedious and time consuming. Fortunately, there is a better way to achieve this using Kubernetes.

Note: these instructions are mildly hacky for now, as we get run once semantics and logging they will get better

There is a testing image `brendanburns/flake` up on the docker hub. We will use this image to test our fix.

Create a replication controller with the following config:

```
id: flakecontroller
kind: ReplicationController
apiVersion: v1beta1
desiredState:
  replicas: 24
  replicaSelector:
    name: flake
  podTemplate:
    desiredState:
      manifest:
        version: v1beta1
        id: ""
        volumes: []
        containers:
        - name: flake
          image: brendanburns/flake
          env:
          - name: TEST_PACKAGE
            value: pkg/tools
          - name: REPO_SPEC
            value: https://github.com/GoogleCloudPlatform/kubernetes
      restartpolicy: {}
    labels:
      name: flake
labels:
  name: flake
```

```
./cluster/kubectl.sh create -f controller.yaml
```

This will spin up 24 instances of the test. They will run to completion, then exit, and the kubelet will restart them, accumulating more and more runs of the test. You can examine the recent runs of the test by calling `docker ps -a` and looking for tasks that exited with non-zero exit codes. Unfortunately, docker ps -a only keeps around the exit status of the last 15-20 containers with the same image, so you have to check them frequently. You can use this script to automate checking for failures, assuming your cluster is running on GCE and has four nodes:

```
echo "" > output.txt
for i in {1..4}; do
  echo "Checking kubernetes-minion-${i}"
  echo "kubernetes-minion-${i}:" >> output.txt
  gcloud compute ssh "kubernetes-minion-${i}" --command="sudo docker ps -a" >> out
done
```

```
grep "Exited ([^0])" output.txt
```

Eventually you will have sufficient runs for your purposes. At that point you can stop and delete the replication controller by running:

```
./cluster/kubectl.sh stop replicationcontroller flakecontroller
```

If you do a final check for flakes with `docker ps -a`, ignore tasks that exited -1, since that's what happens when you stop the replication controller.

Happy flake hunting!

Logging Conventions

The following conventions for the glog levels to use. glog is globally prefered to log for better runtime control.

- glog.Errorf() - Always an error
- glog.Warningf() - Something unexpected, but probably not an error
- glog.Infof() has multiple levels:
- glog.V(0) - Generally useful for this to ALWAYS be visible to an operator
    - Programmer errors
    - Logging extra info about a panic
    - CLI argument handling
- glog.V(1) - A reasonable default log level if you don't want verbosity.
    - Information about config (listening on X, watching Y)
    - Errors that repeat frequently that relate to conditions that can be corrected (pod detected as unhealthy)
- glog.V(2) - Useful steady state information about the service and important log messages that may correlate to significant changes in the system. This is the recommended default log level for most systems.
    - Logging HTTP requests and their exit code
    - System state changing (killing pod)
    - Controller state change events (starting pods)
    - Scheduler log messages
- glog.V(3) - Extended information about changes
    - More info about system state changes
- glog.V(4) - Debug level verbosity (for now)
    - Logging in particularly thorny parts of code where you may want to come back later and check it

As per the comments, the practical default level is V(2). Developers and QE environments may wish to run at V(3) or V(4). If you wish to change the log level, you can pass in `-v=X` where X is the desired maximum level to log.

Releasing Kubernetes

This document explains how to create a Kubernetes release (as in version) and how the version information gets embedded into the built binaries.

Origin of the Sources

Kubernetes may be built from either a git tree (using `hack/build-go.sh`) or from a tarball (using either `hack/build-go.sh` or `go install`) or directly by the Go native build system (using `go get`).

When building from git, we want to be able to insert specific information about the build tree at build time. In particular, we want to use the output of `git describe` to generate the version of Kubernetes and the status of the build tree (add a `-dirty` prefix if the tree was modified.)

When building from a tarball or using the Go build system, we will not have access to the information about the git tree, but we still want to be able to tell whether this build corresponds to an exact release (e.g. v0.3) or is between releases (e.g. at some point in development between v0.3 and v0.4).

Version Number Format

In order to account for these use cases, there are some specific formats that may end up representing the Kubernetes version. Here are a few examples:

- **v0.5**: This is official version 0.5 and this version will only be used when building from a clean git tree at the v0.5 git tag, or from a tree extracted from the tarball corresponding to that specific release.
- **v0.5-15-g0123abcd4567**: This is the `git describe` output and it indicates that we are 15 commits past the v0.5 release and that the SHA1 of the commit where the binaries were built was `0123abcd4567`. It is only possible to have this level of detail in the version information when building from git, not when building from a tarball.
- **v0.5-15-g0123abcd4567-dirty** or **v0.5-dirty**: The extra `-dirty` prefix means that the tree had local modifications or untracked files at the time of the build, so there's no guarantee that the source code matches exactly the state of the tree at the `0123abcd4567` commit or at the `v0.5` git tag (resp.)
- **v0.5-dev**: This means we are building from a tarball or using `go get` or, if we have a git tree, we are using `go install` directly, so it is not possible to inject the git version into the build information. Additionally, this is not an official release, so the `-dev` prefix indicates that the version we are building is after `v0.5` but before `v0.6`. (There is actually an exception where a commit with `v0.5-dev` is not present on `v0.6`, see later for details.)

Injecting Version into Binaries

In order to cover the different build cases, we start by providing information that can be used when using only Go build tools or when we do not have the git version information available.

To be able to provide a meaningful version in those cases, we set the contents of variables in a Go source file that will be used when no overrides are present.

We are using `pkg/version/base.go` as the source of versioning in absence of information from git. Here is a sample of that file's contents:

```
var (
    gitVersion   string = "v0.4-dev"  // version from git, output of $(git descr
    gitCommit    string = ""          // sha1 from git, output of $(git rev-pars
)
```

This means a build with `go install` or `go get` or a build from a tarball will yield binaries that will identify themselves as `v0.4-dev` and will not be able to provide you with a SHA1.

To add the extra versioning information when building from git, the `hack/build-go.sh` script will gather that information (using `git describe` and `git rev-parse`) and then create a `-ldflags` string to pass to `go install` and tell the Go linker to override the contents of those variables at build time. It can, for instance, tell it to override `gitVersion` and set it to `v0.4-13-g4567bcdef6789-dirty` and set `gitCommit` to `4567bcdef6789...` which is the complete SHA1 of the (dirty) tree used at build time.

Handling Official Versions

Handling official versions from git is easy, as long as there is an annotated git tag pointing to a specific version then `git describe` will return that tag exactly which will match the idea of an official version (e.g. `v0.5`).

Handling it on tarballs is a bit harder since the exact version string must be present in `pkg/version/base.go` for it to get embedded into the binaries. But simply creating a commit with `v0.5` on its own would mean that the commits coming after it would also get the `v0.5` version when built from tarball or `go get` while in fact they do not match `v0.5` (the one that was tagged) exactly.

To handle that case, creating a new release should involve creating two adjacent commits where the first of them will set the version to `v0.5` and the second will set it to `v0.5-dev`. In that case, even in the presence of merges, there will be a single comit where the exact `v0.5` version will be used and all others around it will either have `v0.4-dev` or `v0.5-dev`.

The diagram below illustrates it.

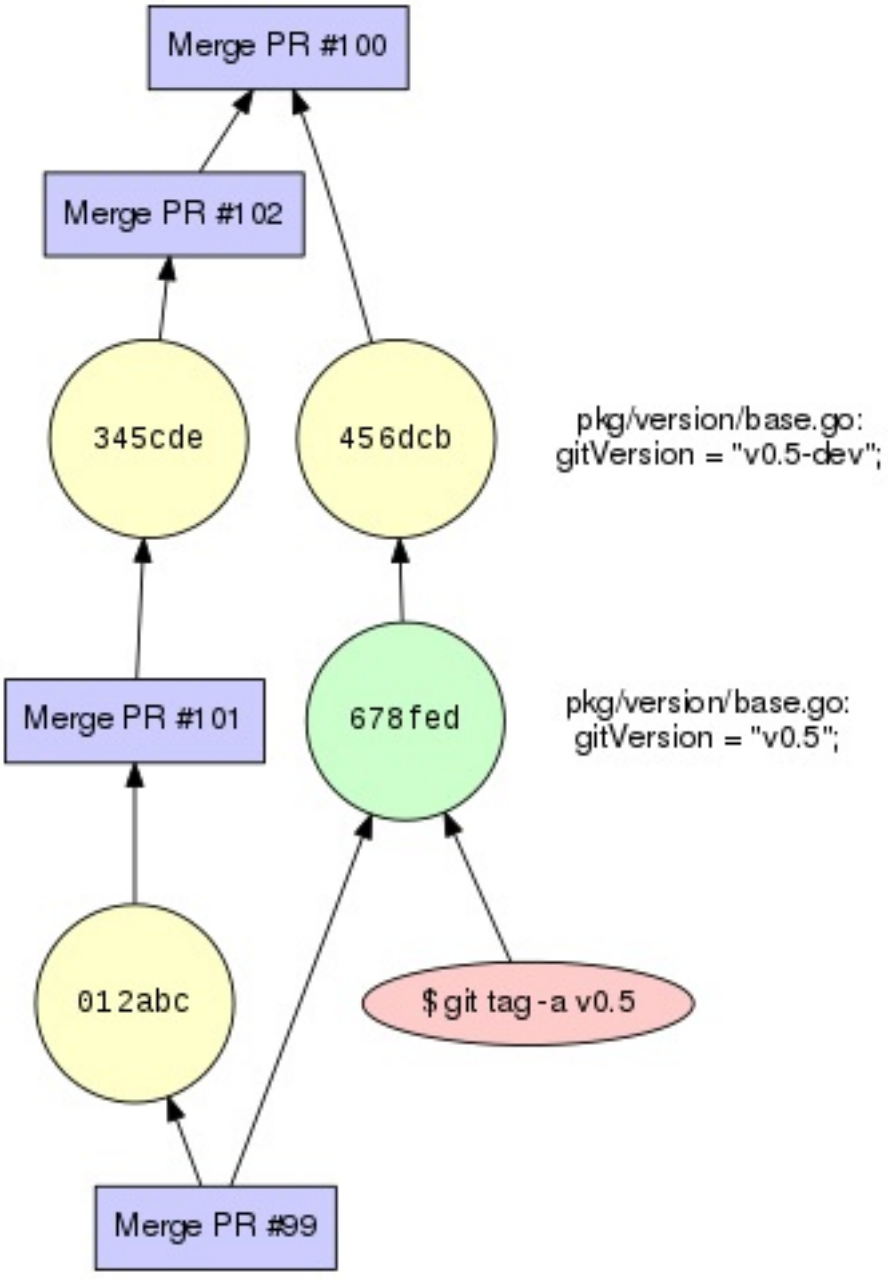


Diagram of git commits involved in the release

After working on `v0.4-dev` and merging PR 99 we decide it is time to release `v0.5`. So we start a new branch, create one commit to update `pkg/version/base.go` to include `gitVersion = "v0.5"` and `git commit` it.

We test it and make sure everything is working as expected.

Before sending a PR for it, we create a second commit on that same branch, updating `pkg/version/base.go` to include `gitVersion = "v0.5-dev"`. That will ensure that further builds (from tarball or `go install`) on that tree will always include the `-dev` prefix and will not have a `v0.5` version (since they do not match the official `v0.5` exactly.)

We then send PR 100 with both commits in it.

Once the PR is accepted, we can use `git tag -a` to create an annotated tag *pointing to the one commit* that has `v0.5` in `pkg/version/base.go` and push it to GitHub. (Unfortunately GitHub tags/releases are not annotated tags, so this needs to be done from a git client and pushed to GitHub using SSH.)

Parallel Commits

While we are working on releasing `v0.5`, other development takes place and other PRs get merged. For instance, in the example above, PRs 101 and 102 get merged to the master branch before the versioning PR gets merged.

This is not a problem, it is only slightly inaccurate that checking out the tree at commit `012abc` or commit `345cde` or at the commit of the merges of PR 101 or 102 will yield a version of `v0.4-dev` *but* those commits are not present in `v0.5`.

In that sense, there is a small window in which commits will get a `v0.4-dev` or `v0.4-N-gXXX` label and while they're indeed later than `v0.4` but they are not really before `v0.5` in that `v0.5` does not contain those commits.

Unfortunately, there is not much we can do about it. On the other hand, other projects seem to live with that and it does not really become a large problem.

As an example, Docker commit a327d9b91edf has a `v1.1.1-N-gXXX` label but it is not present in Docker `v1.2.0`:

```
$ git describe a327d9b91edf
v1.1.1-822-ga327d9b91edf

$ git log --oneline v1.2.0..a327d9b91edf
a327d9b91edf Fix data space reporting from Kb/Mb to KB/MB

(Non-empty output here means the commit is not present on v1.2.0.)
```

Getting started on Microsoft Azure

Azure Prerequisites

1. You need an Azure account. Visit http://azure.microsoft.com/ to get started.
2. Install and configure the Azure cross-platform command-line interface. http://azure.microsoft.com/en-us/documentation/articles/xplat-cli/
3. Make sure you have a default account set in the Azure cli, using `azure account set`

Prerequisites for your workstation

1. Be running a Linux or Mac OS X.
2. Get or build a binary release
3. If you want to build your own release, you need to have Docker installed. On Mac OS X you can use boot2docker.

Setup

The cluster setup scripts can setup Kubernetes for multiple targets. First modify `cluster/kube-env.sh` to specify azure:

```
KUBERNETES_PROVIDER="azure"
```

Next, specify an existing virtual network in `cluster/azure/config-defualt.sh`:

```
AZ_VNET=<vnet name>
```

Now you're ready.

You can then use the `cluster/kube-*.sh` scripts to manage your azure cluster, start with:

```
cluster/kube-up.sh
```

The script above will start (by default) a single master VM along with 4 worker VMs. You can tweak some of these parameters by editing `cluster/azure/config-default.sh`.

Running a container (simple version)

Once you have your instances up and running, the `hack/build-go.sh` script sets up your Go workspace and builds the Go components.

The `kubectl.sh` line below spins up two containers running Nginx running on port 80:

```
cluster/kubectl.sh run-container my-nginx --image=dockerfile/nginx --replicas=2 --
```

To stop the containers:

```
cluster/kubectl.sh stop rc my-nginx
```

To delete the containers:

```
cluster/kubectl.sh delete rc my-nginx
```

Running a container (more complete version)

You can create a pod like this:

```
cd kubernetes
cluster/kubectl.sh create -f api/examples/pod.json
```

Where pod.json contains something like:

```
{
  "id": "php",
  "kind": "Pod",
  "apiVersion": "v1beta1",
  "desiredState": {
    "manifest": {
      "version": "v1beta1",
      "id": "php",
      "containers": [{
        "name": "nginx",
        "image": "dockerfile/nginx",
        "ports": [{
          "containerPort": 80,
          "hostPort": 8080
        }],
        "livenessProbe": {
          "enabled": true,
          "type": "http",
          "initialDelaySeconds": 30,
          "httpGet": {
            "path": "/index.html",
            "port": "8080"
          }
        }
      }]
    }
  },
  "labels": {
    "name": "foo"
  }
}
```

You can see your cluster's pods:

```
cluster/kubectl.sh get pods
```

and delete the pod you just created:

```
cluster/kubectl.sh delete pods php
```

Look in `api/examples/` for more examples

Tearing down the cluster

```
cluster/kube-down.sh
```

CoreOS - Single Node Kubernetes Cluster

Use the standalone.yaml cloud-config to provision a single node Kubernetes cluster.

CoreOS image versions

AWS

```
aws ec2 create-security-group --group-name kubernetes --description "Kubernetes Se
aws ec2 authorize-security-group-ingress --group-name kubernetes --protocol tcp --
aws ec2 authorize-security-group-ingress --group-name kubernetes --source-security
```

Attention: Replace `<ami_image_id>` bellow for a suitable version of CoreOS image for AWS.

```
aws ec2 run-instances \
--image-id <ami_image_id> \
--key-name <keypair> \
--region us-west-2 \
--security-groups kubernetes \
--instance-type m3.medium \
--user-data file://standalone.yaml
```

GCE

Attention: Replace `<gce_image_id>` bellow for a suitable version of CoreOS image for GCE.

```
gcloud compute instances create standalone \
--image-project coreos-cloud \
--image <gce_image_id> \
--boot-disk-size 200GB \
--machine-type n1-standard-1 \
--zone us-central1-a \
--metadata-from-file user-data=standalone.yaml
```

VMware Fusion

Create a config-drive ISO.

```
mkdir -p /tmp/new-drive/openstack/latest/
cp standalone.yaml /tmp/new-drive/openstack/latest/user_data
hdiutil makehybrid -iso -joliet -joliet-volume-name "config-2" -joliet -o standalo
```

Boot the vmware image using the `standalone.iso` as a config drive.

Getting started on CoreOS

There are multiple guides on running Kubernetes with CoreOS:

- Single Node Cluster
- Multi-node Cluster
- Setup Multi-node Cluster on GCE in an easy way from OS X
- Multi-node cluster using cloud-config and Weave on Vagrant
- Multi-node cluster using cloud-config and Vagrant
- Multi-node cluster with Vagrant and fleet units using a small OS X App

Configuring kubernetes on Fedora via Ansible.

Configuring kubernetes on Fedora via Ansible offers a simple way to quickly create a clustered environment with little effort.

Requirements:

1. Host able to run ansible and able to clone the following repo: kubernetes-ansible
2. A Fedora 20+ or RHEL7 host to act as cluster master
3. As many Fedora 20+ or RHEL7 hosts as you would like, that act as cluster minions

The hosts can be virtual or bare metal. The only requirement to make the ansible network setup work is that all of the machines are connected via the same layer 2 network.

Ansible will take care of the rest of the configuration for you - configuring networking, installing packages, handling the firewall, etc... This example will use one master and two minions.

Configuring cluster information:

Hosts:

```
fed1 (master) = 192.168.121.205
fed2 (minion) = 192.168.121.84
fed3 (minion) = 192.168.121.116
```

Make sure your local machine has ansible installed

```
yum install -y ansible
```

Clone the kubernetes-ansible repo on the host running Ansible.

```
git clone https://github.com/eparis/kubernetes-ansible.git
cd kubernetes-ansible
```

Tell ansible about each machine and its role in your cluster.

Get the IP addresses from the master and minions. Add those to the inventory file at the root of the repo on the host running Ansible. Ignore the kube_ip_addr= option for a moment.

```
[masters]
192.168.121.205

[etcd]
192.168.121.205

[minions]
192.168.121.84  kube_ip_addr=[ignored]
192.168.121.116 kube_ip_addr=[ignored]
```

Tell ansible which user has ssh access (and sudo access to root)

edit: group_vars/all.yml

```
ansible_ssh_user: root
```

Configuring ssh access to the cluster

If you already have ssh access to every machine using ssh public keys you may skip to configuring the network

Create a password file.

The password file should contain the root password for every machine in the cluster. It will be used in order to lay down your ssh public key. Make sure your machines sshd-config allows password logins from root.

```
echo "password" > ~/rootpassword
```

Agree to accept each machine's ssh public key

```
ansible-playbook -i inventory ping.yml # This will look like it fails, that's ok
```

Push your ssh public key to every machine

```
ansible-playbook -i inventory keys.yml
```

Configuring the network

If you already have configured your network and docker will use it correctly, skip to setting up the cluster

The ansible scripts are quite hacky configuring the network, see the README

Configure the ip addresses which should be used to run pods on each machine

The IP address pool used to assign addresses to pods for each minion is the kube_ip_addr= option. Choose a /24 to use for each minion and add that to you inventory file.

```
[minions]
192.168.121.84  kube_ip_addr=10.0.1.0
192.168.121.116 kube_ip_addr=10.0.2.0
```

Run the network setup playbook

```
ansible-playbook -i inventory hack-network.yml
```

Setting up the cluster

Configure the IP addresses used for services

Each kubernetes service gets its own IP address. These are not real IPs. You need only select a range of IPs which are not in use elsewhere in your environment. This must be done even if you do not use the network setup provided by the ansible scripts.

edit: group_vars/all.yml

```
kube_service_addresses: 10.254.0.0/16
```

Tell ansible to get to work!

```
ansible-playbook -i inventory setup.yml
```

Testing and using your new cluster

That's all there is to it. It's really that easy. At this point you should have a functioning kubernetes cluster.

Show services running on masters and minions.

```
systemctl | grep -i kube
```

Show firewall rules on the masters and minions.

```
iptables -nvL
```

Create the following apache.json file and deploy pod to minion.

```
cat ~/apache.json
{
  "id": "fedoraapache",
  "kind": "Pod",
  "apiVersion": "v1beta1",
  "desiredState": {
    "manifest": {
      "version": "v1beta1",
      "id": "fedoraapache",
      "containers": [{
        "name": "fedoraapache",
        "image": "fedora/apache",
        "ports": [{
          "containerPort": 80,
          "hostPort": 80
        }]
      }]
    }
  },
  "labels": {
    "name": "fedoraapache"
  }
}

/usr/bin/kubectl create -f apache.json
```

Check where the pod was created

```
/usr/bin/kubectl get pod fedoraapache
```

Check Docker status on minion.

```
docker ps
docker images
```

After the pod is 'Running' Check web server access on the minion

```
curl http://localhost
```

Getting started on Fedora

This is a getting started guide for Fedora. It is a manual configuration so you understand all the underlying packages / services / ports, etc...

This guide will only get ONE minion working. Multiple minions requires a functional networking configuration done outside of kubernetes. Although the additional kubernetes configuration requirements should be obvious.

The kubernetes package provides a few services: kube-apiserver, kube-scheduler, kube-controller-manager, kubelet, kube-proxy. These services are managed by systemd and the configuration resides in a central location: /etc/kubernetes. We will break the services up between the hosts. The first host, fed-master, will be the kubernetes master. This host will run the kube-apiserver, kube-controller-manager, and kube-scheduler. In addition, the master will also run *etcd*. The remaining host, fed-minion will be the minion and run kubelet, proxy, cadvisor and docker.

System Information:

Hosts:

```
fed-master = 192.168.121.9
fed-minion = 192.168.121.65
```

Prepare the hosts:

- Install kubernetes on all hosts - fed-{master,minion}. This will also pull in etcd, docker, and cadvisor.

```
yum -y install --enablerepo=updates-testing kubernetes
```

- Add master and minion to /etc/hosts on all machines (not needed if hostnames already in DNS)

```
echo "192.168.121.9 fed-master
192.168.121.65  fed-minion" >> /etc/hosts
```

- Edit /etc/kubernetes/config which will be the same on all hosts to contain:

```
# Comma seperated list of nodes in the etcd cluster
KUBE_ETCD_SERVERS="--etcd_servers=http://fed-master:4001"

# logging to stderr means we get it in the systemd journal
KUBE_LOGTOSTDERR="--logtostderr=true"

# journal message level, 0 is debug
KUBE_LOG_LEVEL="--v=0"

# Should this cluster be allowed to run privleged docker containers
KUBE_ALLOW_PRIV="--allow_privileged=false"
```

- Disable the firewall on both the master and minon, as docker does not play well with other firewall rule managers

```
systemctl disable iptables-services firewalld
systemctl stop iptables-services firewalld
```

Configure the kubernetes services on the master.

- Edit /etc/kubernetes/apiserver to appear as such:

```
# The address on the local server to listen to.
KUBE_API_ADDRESS="--address=0.0.0.0"
```

```
# The port on the local server to listen on.
KUBE_API_PORT="--port=8080"

# How the replication controller and scheduler find the kube-apiserver
KUBE_MASTER="--master=http://fed-master:8080"

# Port minions listen on
KUBELET_PORT="--kubelet_port=10250"

# Address range to use for services
KUBE_SERVICE_ADDRESSES="--portal_net=10.254.0.0/16"

# Add you own!
KUBE_API_ARGS=""
```

- Edit /etc/kubernetes/controller-manager to appear as such:

```
  # Comma seperated list of minions
  KUBELET_ADDRESSES="--machines=fed-minion"
```

- Start the appropriate services on master:

```
for SERVICES in etcd kube-apiserver kube-controller-manager kube-scheduler; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

Configure the kubernetes services on the minion.

We need to configure the kubelet and start the kubelet and proxy

- Edit /etc/kubernetes/kubelet to appear as such:

```
# The address for the info server to serve on
KUBELET_ADDRESS="--address=0.0.0.0"

# The port for the info server to serve on
KUBELET_PORT="--port=10250"

# You may leave this blank to use the actual hostname
KUBELET_HOSTNAME="--hostname_override=fed-minion"

# Add your own!
KUBELET_ARGS=""
```

- Start the appropriate services on minion (fed-minion).

```
for SERVICES in kube-proxy kubelet docker; do
    systemctl restart $SERVICES
    systemctl enable $SERVICES
    systemctl status $SERVICES
done
```

You should be finished!

- Check to make sure the cluster can see the minion (on fed-master)

```
kubectl get minions
NAME                LABELS
fed-minion          <none>
```

The cluster should be running! Launch a test pod.

You should have a functional cluster, check out 101!

Getting started locally

Requirements

Linux

Not running Linux? Consider running Linux in a local virtual machine with Vagrant, or on a cloud provider like Google Compute Engine

Docker

At least Docker 1.3+. Ensure the Docker daemon is running and can be contacted (try `docker ps`). Some of the kubernetes components need to run as root, which normally works fine with docker.

etcd

You need an etcd in your path, please make sure it is installed and in your `$PATH`.

go

You need go in your path, please make sure it is installed and in your `$PATH`.

Starting the cluster

In a separate tab of your terminal, run the following (since one needs sudo access to start/stop kubernetes daemons, it is easier to run the entire script as root):

```
cd kubernetes
hack/local-up-cluster.sh
```

This will build and start a lightweight local cluster, consisting of a master and a single minion. Type Control-C to shut it down.

You can use the cluster/kubectl.sh script to interact with the local cluster. hack/local-up-cluster.sh will print the commands to run to point kubectl at the local cluster.

Running a container

Your cluster is running, and you want to start running containers!

You can now use any of the cluster/kubectl.sh commands to interact with your local setup.

```
cluster/kubectl.sh get pods
cluster/kubectl.sh get services
cluster/kubectl.sh get replicationControllers
cluster/kubectl.sh run-container my-nginx --image=dockerfile/nginx --replicas=2 --


## begin wait for provision to complete, you can monitor the docker pull by openi
  sudo docker images
  ## you should see it pulling the dockerfile/nginx image, once the above command
  sudo docker ps
  ## you should see your container running!
  exit
## end wait

## introspect kubernetes!
cluster/kubectl.sh get pods
```

```
cluster/kubectl.sh get services
cluster/kubectl.sh get replicationControllers
```

Running a user defined pod

Note the difference between a container and a pod. Since you only asked for the former, kubernetes will create a wrapper pod for you. However you can't view the nginx start page on localhost. To verify that nginx is running you need to run `curl` within the docker container (try `docker exec`).

You can control the specifications of a pod via a user defined manifest, and reach nginx through your browser on the port specified therein:

```
cluster/kubectl.sh create -f api/examples/pod.json
```

Congratulations!

Troubleshooting

I can't reach service IPs on the network.

Some firewall software that uses iptables may not interact well with kubernetes. If you're having trouble around networking, try disabling any firewall or other iptables-using systems, first.

By default the IP range for service portals is 10.0.. - depending on your docker installation, this may conflict with IPs for containers. If you find containers running with IPs in this range, edit hack/local-cluster-up.sh and change the portal_net flag to something else.

I cannot create a replication controller with replica size greater than 1! What gives?

You are running a single minion setup. This has the limitation of only supporting a single replica of a given pod. If you are interested in running with larger replica sizes, we encourage you to try the local vagrant setup or one of the cloud providers.

I changed Kubernetes code, how do I run it?

```
cd kubernetes
hack/build-go.sh
hack/local-up-cluster.sh
```

kubectl claims to start a container but `get pods` and `docker ps` don't show it.

One or more of the kubernetes daemons might've crashed. Tail the logs of each in /tmp.

Logging

Experimental work in progress.

Logging with Fluentd and Elastiscsearch

To enable logging of the stdout and stderr output of every Docker container in a Kubernetes cluster set the shell environment variables `ENABLE_NODE_LOGGING` to `true` and `LOGGING_DESTINATION` to `elasticsearch`.

e.g. in bash:

```
export ENABLE_NODE_LOGGING=true
export LOGGING_DESTINATION=elasticsearch
```

This will instantiate a Fluentd instance on each node which will collect all the Dcoker container log files. The collected logs will be targetted at an Elasticsearch instance assumed to be running on the local node and accepting log information on port 9200. This can be accomplished by writing a pod specification and service specification to define an Elasticsearch service (more informaiton to follow shortly in the contrib directory).

Logging with Fluentd and Google Compute Platform

To enable logging of Docker contains in a cluster using Google Compute Platform set the config flags `ENABLE_NODE_LOGGING` to `true` and `LOGGING_DESTINATION` to `gcp`.

Rackspace

In general, the dev-build-and-up.sh workflow for Rackspace is the similar to GCE. The specific implementation is different due to the use of CoreOS, Rackspace Cloud Files and network design.

These scripts should be used to deploy development environments for Kubernetes. If your account leverages RackConnect or non-standard networking, these scripts will most likely not work without modification.

NOTE: The rackspace scripts do NOT rely on `saltstack`.

The current cluster design is inspired by: - corekube - Angus Lees

Prerequisites

1. You need to have both `nova` and `swiftly` installed. It's recommended to use a python virtualenv to install these packages into.
2. Make sure you have the appropriate environment variables set to interact with the OpenStack APIs. See Rackspace Documentation for more details.

Provider: Rackspace

- To use Rackspace as the provider, set the KUBERNETES_PROVIDER ENV variable: `export KUBERNETES_PROVIDER=rackspace` and run the `bash hack/dev-build-and-up.sh` script.

Build

1. The kubernetes binaries will be built via the common build scripts in `build/`.
2. If you've set the ENV `KUBERNETES_PROVIDER=rackspace`, the scripts will upload `kubernetes-server-linux-amd64.tar.gz` to Cloud Files.
3. A cloud files container will be created via the `swiftly` CLI and a temp URL will be enabled on the object.
4. The built `kubernetes-server-linux-amd64.tar.gz` will be uploaded to this container and the URL will be passed to master/minions nodes when booted.

Cluster

1. There is a specific `cluster/rackspace` directory with the scripts for the following steps:
2. A cloud network will be created and all instances will be attached to this network. We will connect the master API and minion kubelet service via this network.
3. A SSH key will be created and uploaded if needed. This key must be used to ssh into the machines since we won't capture the password.
4. A master and minions will be created via the `nova` CLI. A `cloud-config.yaml` is generated and provided as user-data with the entire configuration for the systems.
5. We then boot as many minions as defined via `$RAX_NUM_MINIONS`.

Some notes:

- The scripts expect `eth2` to be the cloud network that the containers will communicate across.
- A number of the items in `config-default.sh` are overridable via environment variables.
- For older versions please either:
- Sync back to `v0.3` with `git checkout v0.3`
- Download a snapshot of `v0.3`

Network Design

- eth0 - Public Interface used for servers/containers to reach the internet
- eth1 - ServiceNet - Intra-cluster communication (k8s, etcd, etc) communicate via this interface. The `cloud-config` files use the special CoreOS identifier `$private_ipv4` to configure the services.
- eth2 - Cloud Network - Used for k8s pods to communicate with one another. The proxy service will pass traffic via this interface.

IaaS Provider | Config. Mgmt | OS | Docs | Support Level | Notes ---------- | ---------- | ------ | ---------- | ---------- | ----- GCE | Saltstack | Debian | docs | Project | Tested with 0.9.2 by @satnam6502 Vagrant | Saltstack | Fedora | docs | Project | Vagrant | custom | Fedora | docs | Project | Uses K8s v0.5-8 Vagrant | Ansible | Fedora | docs | Project | Uses K8s v0.5-8 GKE | | | docs | Commercial | Uses K8s version 0.9.2 AWS | CoreOS | CoreOS | docs | Community | Uses K8s version 0.10.1 GCE | CoreOS | CoreOS | docs | Community (@kelseyhightower) | Uses K8s version 0.10.1 Vagrant | CoreOS | CoreOS | docs | Community (@pires) | Uses K8s version 0.10.1 CloudStack | Ansible | CoreOS | docs| Community (@sebgoa) | Uses K8s version 0.9.1 Vmware | | Debian | docs | Community (@pietern) | Uses K8s version 0.9.1 AWS | Saltstack | Ubuntu | docs | Community (@justinsb) | Uses K8s version 0.5.0 Vmware | CoreOS | CoreOS | docs | Community (@kelseyhightower) | Azure | Saltstack | Ubuntu | docs | Community (@jeffmendoza) | Bare-metal | custom | Ubuntu | docs | Community (@jainvipin) | Local | | | docs | Inactive | Ovirt | | | docs | Inactive | Rackspace | CoreOS | CoreOS | docs | Inactive | Bare-metal | custom | CentOS | docs | Community(@coolsvap) | Uses K8s v0.9.1 Definition of columns: - **IaaS Provider** is who/what provides the virtual or physical machines (nodes) that Kubernetes runs on. - **OS** is the base operating system of the nodes. - **Config. Mgmt** is the configuration management system that helps install and maintain kubernetes software on the nodes. - Support Levels - **Project**: Kubernetes Committers regularly use this configuration, so it usually works with the latest release of Kubernetes. - **Commercial**: A commercial offering with its own support arrangements. - **Community**: Actively supported by community contributions. May not work with more recent releases of kubernetes. - **Inactive**: No active maintainer. Not recommended for first-time K8s users, and may be deleted soon.

Getting started on Ubuntu

This document describes how to get started to run kubernetes services on a single host (which is acting both as master and minion) for ubuntu systems. It consists of three steps

1. Make kubernetes and etcd binaries
2. Install upstart scripts
3. Customizing ubuntu launch

1. Make kubernetes and etcd binaries

Either build or download the latest [kubernetes binaries] (https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/getting-started-guides/binary_release.html)

Copy the kube binaries into `/opt/bin` or a path of your choice

Similarly pull an `etcd` binary from etcd releases or build the `etcd` yourself using instructions at https://github.com/coreos/etcd

Copy the `etcd` binary into `/opt/bin` or path of your choice

2. Install upstart scripts

Running ubuntu/util.sh would install/copy the scripts for upstart to pick up. The script may warn you on some valid problems/conditions

```
$ cd kubernetes/cluster/ubuntu
$ sudo ./util.sh
```

After this the kubernetes and `etcd` services would be up and running. You can use `service start/stop/restart/force-reload` on the services.

Launching and scheduling containers using kubectl can also be used at this point, as explained mentioned in the examples

3. Customizing the ubuntu launch

To customize the defaults you will need to tweak `/etc/default/kube*` files and restart the appropriate services. This is needed if the binaries are copied in a place other than `/opt/bin`. A run could look like

```
$ sudo cat /etc/default/etcd
# Etcd Upstart and SysVinit configuration file

# Customize etcd location
# ETCD="/opt/bin/etcd"

# Use ETCD_OPTS to modify the start/restart options
ETCD_OPTS="-listen-client-urls=http://127.0.0.1:4001"

# Add more environment settings used by etcd here

$ sudo service etcd status
etcd start/running, process 834
$ sudo service etcd restart
etcd stop/waiting
etcd start/running, process 29050
```

Images

Each container in a pod has its own image. Currently, the only type of image supported is a Docker Image.

You create your Docker image and push it to a registry before referring to it in a kubernetes pod.

The `image` property of a container supports the same syntax as the `docker` command does, including private registries and tags.

Using a Private Registry

Google Container Registry

Kubernetes has native support for the Google Container Regisry, when running on Google Compute Engine. If you are running your cluster on Google Compute Engine or Google Container Engine, simply use the full image name (e.g. gcr.io/my_project/image:tag) and the kubelet will automatically authenticate and pull down your private image.

Other Private Registries

Docker stores keys for private registries in a `.dockercfg` file. Create a config file by running `docker login <registry>.<domain>` and then copying the resulting `.dockercfg` file to the kubelet working dir. The kubelet working dir varies by cloud provider. It is `/` on GCE and `/home/core` on CoreOS. You can determine the working dir by running this command:
`sudo ls -ld /proc/$(pidof kubelet)/cwd` on a kNode.

All users of the cluster will have access to any private registry in the `.dockercfg`.

Preloading Images

Be default, the kubelet will try to pull each image from the specified registry. However, if the `imagePullPolicy` property of the container is set to `IfNotPresent` or `Never`, then a local image is used (preferentially or exclusively, respectively).

This can be used to preload certain images for speed or as an alternative to authenticating to a private registry.

Pull Policy is per-container, but any user of the cluster will have access to all local images.

.kubeconfig files

In order to easily switch between multiple clusters, a .kubeconfig file was defined. This file contains a series of authentication mechanisms and cluster connection information associated with nicknames. It also introduces the concept of a tuple of authentication information (user) and cluster connection information called a context that is also associated with a nickname.

Multiple files are .kubeconfig files are allowed. At runtime they are loaded and merged together along with override options specified from the command line (see rules below).

Related discussion

https://github.com/GoogleCloudPlatform/kubernetes/issues/1755

Example .kubeconfig file

```
apiVersion: v1
clusters:
- cluster:
    api-version: v1beta1
    server: http://cow.org:8080
  name: cow-cluster
- cluster:
    certificate-authority: path/to/my/cafile
    server: https://horse.org:4443
  name: horse-cluster
- cluster:
    insecure-skip-tls-verify: true
    server: https://pig.org:443
  name: pig-cluster
contexts:
- context:
    cluster: horse-cluster
    namespace: chisel-ns
    user: green-user
  name: federal-context
- context:
    cluster: pig-cluster
    namespace: saw-ns
    user: black-user
  name: queen-anne-context
current-context: federal-context
kind: Config
preferences:
  colors: true
users:
- name: black-user
  user:
    auth-path: path/to/my/existing/.kubernetes_auth_file
- name: blue-user
  user:
    token: blue-token
- name: green-user
  user:
    client-certificate: path/to/my/client/cert
    client-key: path/to/my/client/key
```

Loading and merging rules

The rules for loading and merging the .kubeconfig files are straightforward, but there are a lot of them. The final config is built in this order: 1. Merge together the kubeconfig itself. This is done with the following hierarchy and merge rules:

```
Empty filenames are ignored.  Files with non-deserializable content produced err
The first file to set a particular value or map key wins and the value or map ke
This means that the first file to set CurrentContext will have its context prese
1.  CommandLineLocation - the value of the `kubeconfig` command line option
1.  EnvVarLocation - the value of $KUBECONFIG
1.  CurrentDirectoryLocation - ``pwd``/.kubeconfig
1.  HomeDirectoryLocation = ~/.kube/.kubeconfig
```

1. Determine the context to use based on the first hit in this chain
    1. command line argument - the value of the `context` command line option
    2. current-context from the merged kubeconfig file
    3. Empty is allowed at this stage
2. Determine the cluster info and user to use. At this point, we may or may not have a context. They are built based on the first hit in this chain. (run it twice, once for user, once for cluster)
    1. command line argument - `user` for user name and `cluster` for cluster name
    2. If context is present, then use the context's value
    3. Empty is allowed
3. Determine the actual cluster info to use. At this point, we may or may not have a cluster info. Build each piece of the cluster info based on the chain (first hit wins):
    1. command line arguments - `server`, `api-version`, `certificate-authority`, and `insecure-skip-tls-verify`
    2. If cluster info is present and a value for the attribute is present, use it.
    3. If you don't have a server location, error.

4. User is build using the same rules as cluster info, EXCEPT that you can only have one authentication technique per user.

    The command line flags are: `auth-path`, `client-certificate`, `client-key`, and `token`. If there are two conflicting techniques, fail.

5. For any information still missing, use default values and potentially prompt for authentication information

Manipulation of .kubeconfig via `kubectl config <subcommand>`

In order to more easily manipulate .kubeconfig files, there are a series of subcommands to `kubectl config` to help.

```
kubectl config set-credentials name --auth-path=path/to/authfile --client-certific
  Sets a user entry in .kubeconfig.  If the referenced name already exists, it wil
kubectl config set-cluster name --server=server --skip-tls=bool --certificate-auth
  Sets a cluster entry in .kubeconfig.  If the referenced name already exists, it
kubectl config set-context name --user=string --cluster=string --namespace=string
  Sets a config entry in .kubeconfig.  If the referenced name already exists, it w
kubectl config use-context name
  Sets current-context to name
kubectl config set property-name property-value
  Sets arbitrary value in .kubeconfig
kubectl config unset property-name
  Unsets arbitrary value in .kubeconfig
kubectl config view --local=true --global=false --kubeconfig=specific/filename --m
  Displays the merged (or not) result of the specified .kubeconfig file

--local, --global, and --kubeconfig are valid flags for all of these operations.
```

Example

```
$kubectl config set-credentials myself --auth-path=path/to/my/existing/auth-file
$kubectl config set-cluster local-server --server=http://localhost:8080
$kubectl config set-context default-context --cluster=local-server --user=myself
$kubectl config use-context default-context
$kubectl config set contexts.default-context.namespace the-right-prefix
$kubectl config view
```

produces this output

```
clusters:
  local-server:
    server: http://localhost:8080
contexts:
  default-context:
    cluster: local-server
    namespace: the-right-prefix
    user: myself
current-context: default-context
preferences: {}
users:
  myself:
    auth-path: path/to/my/existing/auth-file
```

and a .kubeconfig file that looks like this

```
apiVersion: v1
clusters:
- cluster:
    server: http://localhost:8080
  name: local-server
contexts:
- context:
    cluster: local-server
    namespace: the-right-prefix
    user: myself
```

```
  name: default-context
current-context: default-context
kind: Config
preferences: {}
users:
- name: myself
  user:
    auth-path: path/to/my/existing/auth-file
```

Commands for the example file

```
$kubectl config set preferences.colors true
$kubectl config set-cluster cow-cluster --server=http://cow.org:8080 --api-version
$kubectl config set-cluster horse-cluster --server=https://horse.org:4443 --certi
$kubectl config set-cluster pig-cluster --server=https://pig.org:443 --insecure-sk
$kubectl config set-credentials black-user --auth-path=path/to/my/existing/.kubern
$kubectl config set-credentials blue-user --token=blue-token
$kubectl config set-credentials green-user --client-certificate=path/to/my/client/
$kubectl config set-context queen-anne-context --cluster=pig-cluster --user=black-
$kubectl config set-context federal-context --cluster=horse-cluster --user=green-u
$kubectl config use-context federal-context
```

KUBERNETES(1) kubernetes User Manuals

Scott Collier

October 2014

NAME

kube-apiserver - Provides the API for kubernetes orchestration.

SYNOPSIS

kube-apiserver OPTIONS

DESCRIPTION

The **kubernetes** API server validates and configures data for 3 types of objects: pods, services, and replicationControllers. Beyond just servicing REST operations, the API Server does two other things as well: 1. Schedules pods to worker nodes. Right now the scheduler is very simple. 2. Synchronize pod information (where they are, what ports they are exposing) with the service configuration.

The the kube-apiserver several options.

OPTIONS

--address="" The address on the local server to listen to. Default 127.0.0.1

--allow_privileged="" If true, allow privileged containers.

--alsologtostderr= log to standard error as well as files. Default is false.

--api_prefix="/api" The prefix for API requests on the server. Default '/api'

--cloud_config="" The path to the cloud provider configuration file. Empty string for no configuration file.

--cloud_provider="" The provider for cloud services. Empty string for no provider.

--cors_allowed_origins=[] List of allowed origins for CORS, comma separated. An allowed origin can be a regular expression to support subdomain matching. If this list is empty CORS will not be enabled.

--etcd_servers=[] List of etcd servers to watch (http://ip:port), comma separated

--log_backtrace_at=:0 when logging hits line file:N, emit a stack trace

--log_dir="" If non-empty, write log files in this directory

--log_flush_frequency=5s Maximum number of seconds between log flushes. Default is 5 seconds.

--logtostderr= log to standard error instead of files. Default is false.

--kubelet_port=10250 The port at which kubelet will be listening on the minions. Default is 10250.

--port=8080 The port to listen on. Default is 8080.

--stderrthreshold=0 logs at or above this threshold go to stderr. Default is 0.

--storage_version="" The version to store resources with. Defaults to server preferred.

--v=0 Log level for V logs.

--version=false Print version information and quit. Default is false.

--vmodule= comma-separated list of pattern=N settings for file-filtered logging

EXAMPLES

```
/usr/bin/kube-apiserver --logtostderr=true --v=0 --etcd_servers=http://127.0.0.1:4
```

HISTORY

October 2014, Originally compiled by Scott Collier (scollier at redhat dot com) based on the kubernetes source material and internal work.

KUBERNETES(1) kubernetes User Manuals

Scott Collier

October 2014

NAME

kube-controller-manager - Enforces kubernetes services.

SYNOPSIS

kube-controller-manager OPTIONS

DESCRIPTION

The **kubernetes** controller manager is really a service that is layered on top of the simple pod API. To enforce this layering, the logic for the replicationController is actually broken out into another server. This server watches etcd for changes to replicationController objects and then uses the public Kubernetes API to implement the replication algorithm.

The kube-controller-manager has several options.

OPTIONS

--address="" The address on the local server to listen to. Default 127.0.0.1.

--allow_privileged="false" If true, allow privileged containers.

--address="127.0.0.1" The address to serve from.

--alsologtostderr=false log to standard error as well as files.

--api_version="" The API version to use when talking to the server.

--cloud_config="" The path to the cloud provider configuration file. Empty string for no configuration file.

--cloud_provider="" The provider for cloud services. Empty string for no provider.

--minion_regexp="" If non empty, and --cloud_provider is specified, a regular expression for matching minion VMs.

--insecure_skip_tls_verify=false If true, the server's certificate will not be checked for validity. This will make your HTTPS connections insecure.

--log_backtrace_at=:0 when logging hits line file:N, emit a stack trace.

--log_dir="" If non-empty, write log files in this directory.

--log_flush_frequency=5s Maximum number of seconds between log flushes.

--logtostderr=false log to standard error instead of files.

--machines=[] List of machines to schedule onto, comma separated.

--sync_nodes=true If true, and --cloud_provider is specified, sync nodes from the cloud provider. Default true.

--master="" The address of the Kubernetes API server.

--node_sync_peroid=10s The period for syncing nodes from cloudprovider.

--port=10252 The port that the controller-manager's http service runs on.

--stderrthreshold=0 logs at or above this threshold go to stderr.

--v=0 log level for V logs.

--version=false Print version information and quit.

--vmodule= comma-separated list of pattern=N settings for file-filtered logging.

EXAMPLES

```
/usr/bin/kube-controller-manager --logtostderr=true --v=0 --master=127.0.0.1:8080
```

HISTORY

October 2014, Originally compiled by Scott Collier (scollier at redhat dot com) based on the kubernetes source material and internal work.

KUBERNETES(1) kubernetes User Manuals

Scott Collier

October 2014

NAME

kube-proxy - Provides network proxy services.

SYNOPSIS

kube-proxy OPTIONS

DESCRIPTION

The **kubernetes** network proxy runs on each node. This reflects services as defined in the Kubernetes API on each node and can do simple TCP stream forwarding or round robin TCP forwarding across a set of backends. Service endpoints are currently found through Docker-links-compatible environment variables specifying ports opened by the service proxy. Currently the user must select a port to expose the service on on the proxy, as well as the container's port to target.

The kube-proxy takes several options.

OPTIONS

--alsologtostderr=false log to standard error as well as files

--api_version="" The API version to use when talking to the server

--bindaddress="0.0.0.0" The address for the proxy server to serve on (set to 0.0.0.0 or "" for all interfaces)

--etcd_servers=[] List of etcd servers to watch (http://ip:port), comma separated (optional)

--insecure_skip_tls_verify=false If true, the server's certificate will not be checked for validity. This will make your HTTPS connections insecure.

--log_backtrace_at=:0 when logging hits line file:N, emit a stack trace

--log_dir="" If non-empty, write log files in this directory

--log_flush_frequency=5s Maximum number of seconds between log flushes

--logtostderr=false log to standard error instead of files

--master="" The address of the Kubernetes API server

--stderrthreshold=0 logs at or above this threshold go to stderr

--v=0 log level for V logs

--version=false Print version information and quit

--vmodule= comma-separated list of pattern=N settings for file-filtered logging

EXAMPLES

```
/usr/bin/kube-proxy --logtostderr=true --v=0 --etcd_servers=http://127.0.0.1:4001
```

HISTORY

October 2014, Originally compiled by Scott Collier (scollier at redhat dot com) based on the kubernetes source material and internal work.

KUBERNETES(1) kubernetes User Manuals

Scott Collier

October 2014

NAME

kube-scheduler - Schedules containers on hosts.

SYNOPSIS

kube-scheduler OPTIONS

DESCRIPTION

The **kubernetes** scheduler is a policy-rich, topology-aware, workload-specific function that significantly impacts availability, performance, and capacity. The scheduler needs to take into account individual and collective resource requirements, quality of service requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, deadlines, and so on. Workload-specific requirements will be exposed through the API as necessary.

The kube-scheduler can take several options.

OPTIONS

--address="127.0.0.1" The address to serve from.

--alsologtostderr=false log to standard error as well as files.

--api_version="" The API version to use when talking to the server.

--insecure_skip_tls_verify=false If true, the server's certificate will not be checked for validity. This will make your HTTPS connections insecure.

--log_backtrace_at=:0 when logging hits line file:N, emit a stack trace.

--log_dir="" If non-empty, write log files in this directory.

--log_flush_frequency=5s Maximum number of seconds between log flushes.

--logtostderr=false log to standard error instead of files.

--master="" The address of the Kubernetes API server.

--port=10251 The port that the scheduler's http service runs on.

--stderrthreshold=0 logs at or above this threshold go to stderr.

--v=0 log level for V logs.

--version=false Print version information and quit.

--vmodule= comma-separated list of pattern=N settings for file-filtered logging.

EXAMPLES

```
/usr/bin/kube-scheduler --logtostderr=true --v=0 --master=127.0.0.1:8080
```

HISTORY

October 2014, Originally compiled by Scott Collier (scollier@redhat.com) based on the kubernetes source material and internal work.

KUBERNETES(1) kubernetes User Manuals

Scott Collier

October 2014

NAME

kubelet - Processes a container manifest so the containers are launched according to how they are described.

SYNOPSIS

kubelet OPTIONS

DESCRIPTION

The **kubernetes** kubelet runs on each node. The Kubelet works in terms of a container manifest. A container manifest is a YAML or JSON file that describes a pod. The Kubelet takes a set of manifests that are provided in various mechanisms and ensures that the containers described in those manifests are started and continue running.

There are 4 ways that a container manifest can be provided to the Kubelet:

```
File Path passed as a flag on the command line. This file is rechecked every 20 se
HTTP endpoint HTTP endpoint passed as a parameter on the command line. This endpo
etcd server The Kubelet will reach out and do a watch on an etcd server. The etcd
HTTP server The kubelet can also listen for HTTP and respond to a simple API (unde
```

OPTIONS

--address="127.0.0.1" The address for the info server to serve on (set to 0.0.0.0 or "" for all interfaces).

--allow_privileged=false If true, allow containers to request privileged mode. [default=false].

--alsologtostderr=false log to standard error as well as files.

--config="" Path to the config file or directory of files.

--docker_endpoint="" If non-empty, use this for the docker endpoint to communicate with.

--enable_server=true Enable the info server.

--etcd_servers=[] List of etcd servers to watch (http://ip:port), comma separated.

--file_check_frequency=20s Duration between checking config files for new data.

--hostname_override="" If non-empty, will use this string as identification instead of the actual hostname.

--http_check_frequency=20s Duration between checking http for new data.

--log_backtrace_at=:0 when logging hits line file:N, emit a stack trace.

--log_dir="" If non-empty, write log files in this directory.

--log_flush_frequency=5s Maximum number of seconds between log flushes.

--logtostderr=false log to standard error instead of files.

--manifest_url="" URL for accessing the container manifest.

--pod_infra_container_image="kubernetes/pause:latest" The image that pod infra containers in each pod will use.

--port=10250 The port for the info server to serve on.

--registry_burst=10 Maximum size of a bursty pulls, temporarily allows pulls to burst to this number, while still not exceeding registry_qps. Only used if --registry_qps > 0.

--registry_qps=0 If > 0, limit registry pull QPS to this value. If 0, unlimited. [default=0.0].

--root_dir="/var/lib/kubelet" Directory path for managing kubelet files (volume mounts,etc).

--stderrthreshold=0 logs at or above this threshold go to stderr.

--sync_frequency=10s Max period between synchronizing running containers and config.

--v=0 log level for V logs.

--version=false Print version information and quit.

--vmodule= comma-separated list of pattern=N settings for file-filtered logging.

EXAMPLES

```
/usr/bin/kubelet --logtostderr=true --v=0 --etcd_servers=http://127.0.0.1:4001 --a
```

HISTORY

October 2014, Originally compiled by Scott Collier (scollier at redhat dot com) based on the kubernetes source material and internal work.

The life of a pod

Updated: 9/22/2014

This document covers the intersection of pod states, the PodStatus type, the life-cycle of a pod, events, restart policies, and replication controllers. It is not an exhaustive document, but an introduction to the topics.

What is PodStatus?

While `PodStatus` represents the state of a pod, it is not intended to form a state machine. `PodStatus` is an observation of the current state of a pod. As such, we discourage people from thinking about "transitions" or "changes" or "future states".

Events

Since `PodStatus` is not a state machine, there are no edges which can be considered the "reason" for the current state. Reasons can be determined by examining the events for the pod. Events that affect containers, e.g. OOM, are reported as pod events.

TODO(@lavalamp) Event design

Controllers and RestartPolicy

The only controller we have today is `ReplicationController`. `ReplicationController` is *only* appropriate for pods with `RestartPolicy = Always`. `ReplicationController` should refuse to instantiate any pod that has a different restart policy.

There is a legitimate need for a controller which keeps pods with other policies alive. Both of the other policies (`OnFailure` and `Never`) eventually terminate, at which point the controller should stop recreating them. Because of this fundamental distinction, let's hypothesize a new controller, called `JobController` for the sake of this document, which can implement this policy.

Container termination

Containers can terminate with one of two statuses: 1. success: The container exited voluntarily with a status code of 0. 1. failure: The container exited with any other status code or signal, or was stopped by the system.

TODO(@dchen1107) Define ContainerStatus like PodStatus

PodStatus values and meanings

The number and meanings of `PodStatus` values are tightly guarded. Other than what is documented here, nothing should be assumed about pods with a given `PodStatus`.

pending

The pod has been accepted by the system, but one or more of the containers has not been started. This includes time before being schedule as well as time spent downloading images over the network, which could take a while.

running

The pod has been bound to a node, and all of the containers have been started. At least one container is still running (or is in the process of restarting).

succeeded

All containers in the pod have terminated in success.

failed

All containers in the pod have terminated, at least one container has terminated in failure.

Pod lifetime

In general, pods which are created do not disappear until someone destroys them. This might be a human or a `ReplicationController`. The only exception to this rule is that pods with a `PodStatus` of `succeeded` or `failed` for more than some duration (determined by the master) will expire and be automatically reaped.

If a node dies or is disconnected from the rest of the cluster, some entity within the system (call it the NodeController for now) is responsible for applying policy (e.g. a timeout) and marking any pods on the lost node as `failed`.

Examples

- Pod is `running`, 1 container, container exits success
    - Log completion event
    - If RestartPolicy is:
    - Always: restart container, pod stays `running`
    - OnFailure: pod becomes `succeeded`
    - Never: pod becomes `succeeded`
- Pod is `running`, 1 container, container exits failure
    - Log failure event
    - If RestartPolicy is:
    - Always: restart container, pod stays `running`
    - OnFailure: restart container, pod stays `running`
    - Never: pod becomes `failed`
- Pod is `running`, 2 containers, container 1 exits failure
    - Log failure event
    - If RestartPolicy is:
    - Always: restart container, pod stays `running`
    - OnFailure: restart container, pod stays `running`
    - Never: pod stays `running`
    - When container 2 exits...
    - Log failure event
    - If RestartPolicy is:
        - Always: restart container, pod stays `running`
        - OnFailure: restart container, pod stays `running`
        - Never: pod becomes `failed`
- Pod is `running`, container becomes OOM
    - Container terminates in failure
    - Log OOM event
    - If RestartPolicy is:
    - Always: restart container, pod stays `running`
    - OnFailure: restart container, pod stays `running`
    - Never: log failure event, pod becomes `failed`
- Pod is `running`, a disk dies
    - All containers are killed
    - Log appropriate event
    - Pod becomes `failed`
    - If running under a controller, pod will be recreated elsewhere
- Pod is `running`, its node is segmented out
    - NodeController waits for timeout
    - NodeController marks pod `failed`
    - If running under a controller, pod will be recreated elsewhere

Kubernetes Documentation

Getting started guides are in getting-started-guides.

There are example files and walkthroughs in the examples folder.

If you're developing Kubernetes, docs are in the devel folder.

Design docs are in design.

API objects are explained at http://kubernetes.io/third_party/swagger-ui/.

Primary concepts

- **Overview** (overview.html): A brief overview of Kubernetes concepts.
- **Nodes** (node.html): A node is a worker machine in Kubernetes.
- **Pods** (pods.html): A pod is a tightly-coupled group of containers with shared volumes.
- **The Life of a Pod** (pod-states.html): Covers the intersection of pod states, the PodStatus type, the life-cycle of a pod, events, restart policies, and replication controllers.
- **Replication Controllers** (replication-controller.html): A replication controller ensures that a specified number of pod "replicas" are running at any one time.
- **Services** (services.html): A Kubernetes service is an abstraction which defines a logical set of pods and a policy by which to access them.
- **Volumes** (volumes.html): A Volume is a directory, possibly with some data in it, which is accessible to a Container.
- **Labels** (labels.html): Labels are key/value pairs that are attached to objects, such as pods. Labels can be used to organize and to select subsets of objects.
- **Accessing the API** (accessing_the_api.html): Ports, IPs, proxies, and firewall rules.
- **Kubernetes Web Interface** (ui.html): Accessing the Kubernetes web user interface.
- **Kubectl Command Line Interface** (kubectl.html): The `kubectl` command line reference.
- **Roadmap** (roadmap.html): The set of supported use cases, features, docs, and patterns that are required before Kubernetes 1.0.
- **Glossary** (glossary.html): Terms and concepts.

Further reading

- **Annotations** (annotations.html): Attaching arbitrary non-identifying metadata.
- **API Conventions** (api-conventions.html): Defining the verbs and resources used in the Kubernetes API.
- **Authentication Plugins** (authentication.html): The current and planned states of authentication tokens.
- **Authorization Plugins** (authorization.html): Authorization applies to all HTTP requests on the main apiserver port. This doc explains the available authorization implementations.
- **API Client Libraries** (client-libraries.html): A list of existing client libraries, both supported and user-contributed.
- **Kubernetes Container Environment** (container-environment.html): Describes the environment for Kubelet managed containers on a Kubernetes node.
- **DNS Integration with SkyDNS** (dns.html): Resolving a DNS name directly to a Kubernetes service.
- **Identifiers** (identifiers.html): Names and UIDs explained.
- **Images** (images.html): Information about container images and private registries.
- **Logging** (logging.html): Pointers to logging info.
- **Namespaces** (namespaces.html): Namespaces help different projects, teams, or customers to share a kubernetes cluster.
- **Networking** (networking.html): Pod networking overview.
- **OpenVSwitch GRE/VxLAN networking** (ovs-networking.html): Using OpenVSwitch to set up networking between pods across Kubernetes nodes.
- **The Kubernetes Resource Model** (resources.html): Provides resource information such as size, type, and quantity to assist in assigning Kubernetes resources appropriately.
- **Using Salt to configure Kubernetes** (salt.html): The Kubernetes cluster can be configured using Salt.

Replication Controller

What is a *replication controller*?

A *replication controller* ensures that a specified number of pod "replicas" are running at any one time. If there are too many, it will kill some. If there are too few, it will start more. As opposed to just creating singleton pods or even creating pods in bulk, a replication controller replaces pods that are deleted or terminated for any reason, such as in the case of node failure. For this reason, we recommend that you use a replication controller even if your application requires only a single pod.

As discussed in life of a pod, `replicationController` is *only* appropriate for pods with `RestartPolicy = Always`. `ReplicationController` should refuse to instantiate any pod that has a different restart policy. As discussed in issue #503, we expect other types of controllers to be added to Kubernetes to handle other types of workloads, such as build/test and batch workloads, in the future.

A replication controller will never terminate on its own, but it isn't expected to be as long-lived as services. Services may be comprised of pods controlled by multiple replication controllers, and it is expected that many replication controllers may be created and destroyed over the lifetime of a service. Both services themselves and their clients should remain oblivious to the replication controllers that maintain the pods of the services.

How does a replication controller work?

Pod template

A replication controller creates new pods from a template, which is currently inline in the `replicationController` object, but which we plan to extract into its own resource #170.

Rather than specifying the current desired state of all replicas, pod templates are like cookie cutters. Once a cookie has been cut, the cookie has no relationship to the cutter. There is no quantum entanglement. Subsequent changes to the template or even switching to a new template has no direct effect on the pods already created. Similarly, pods created by a replication controller may subsequently be updated directly. This is in deliberate contrast to pods, which do specify the current desired state of all containers belonging to the pod. This approach radically simplifies system semantics and increases the flexibility of the primitive, as demonstrated by the use cases explained below.

Pods created by a replication controller are intended to be fungible and semantically identical, though their configurations may become heterogeneous over time. This is an obvious fit for replicated stateless servers, but replication controllers can also be used to maintain availability of master-elected, sharded, and worker-pool applications. Such applications should use dynamic work assignment mechanisms, such as the etcd lock module or RabbitMQ work queues, as opposed to static/one-time customization of the configuration of each pod, which is considered an anti-pattern. Any pod customization performed, such as vertical auto-sizing of resources (e.g., cpu or memory), should be performed by another online controller process, not unlike the replication controller itself.

Labels

The population of pods that a `replicationController` is monitoring is defined with a label selector, which creates a loosely coupled relationship between the controller and the pods controlled, in contrast to pods, which are more tightly coupled. We deliberately chose not to represent the set of pods controlled using a fixed-length array of pod specifications, because our experience is that that approach increases complexity of management operations, for both clients and the system.

The replication controller should verify that the pods created from the specified template have labels that match its label selector. Though it isn't verified yet, you should also ensure that only one replication controller controls any given pod, by ensuring that the label selectors of replication controllers do not target overlapping sets.

Note that `replicationControllers` may themselves have labels and would generally carry the labels their corresponding pods have in common, but these labels do not affect the behavior of the replication controllers.

Pods may be removed from a replication controller's target set by changing their labels. This technique may be used to remove pods from service for debugging, data recovery, etc. Pods that are removed in this way will be replaced automatically (assuming that the number of replicas is not also changed).

Similarly, deleting a replication controller does not affect the pods it created. It's `replicas` field must first be set to 0 in order to delete the pods controlled. In the future, we may provide a feature to do this and the deletion in a single client operation.

Responsibilities of the replication controller

The replication controller simply ensures that the desired number of pods matches its label selector and are operational. Currently, only terminated pods are excluded from its count. In the future, readiness and other information available from the system may be taken into account, we may add more controls over the replacement policy, and we plan to emit events that could be used by external clients to implement arbitrarily sophisticated replacement and/or scale-down policies.

The replication controller is forever constrained to this narrow responsibility. It itself will not perform readiness nor liveness probes. Rather than performing auto-scaling, it is intended to be controlled by an external auto-scaler (as discussed in #492), which would change its `replicas` field. We will not add scheduling policies (e.g., spreading) to replication controller. Nor should it verify that the pods controlled match the currently specified template, as that would obstruct auto-sizing and other automated processes. Similarly, completion deadlines, ordering dependencies, configuration expansion, and other features belong elsehwere. We even plan to factor out the mechanism for bulk pod creation (#170).

The replication controller is intended to be a composable building-block primitive. We expect higher-level APIs and/or tools to be built on top of it and other complementary primitives for user convenience in the future. The "macro" operations currently supported by kubectl (run-container, stop, resize, rollingupdate) are proof-of-concept examples of this. For instance, we could imagine something like Asgard managing replication controllers, auto-scalers, services, scheduling policies, canaries, etc.

Common usage patterns

Rescheduling

As mentioned above, whether you have 1 pod you want to keep running, or 1000, replication controller will ensure that the specified number of pods exists, even in the event of node failure or pod termination (e.g., due to an action by another control agent).

Scaling

Replication controller makes it easy to scale the number of replicas up or down, either manually or by an auto-scaling control agent, by simply updating the `replicas` field.

Rolling updates

Replication controller is designed to facilitate rolling updates to a service by replacing pods one-by-one.

As explained in #1353, the recommended approach is to create a new replication controller with 1 replica, resize the new (+1) and old (-1) controllers one by one, and then delete the old controller after it reaches 0 replicas. This predictably updates the set of pods regardless of unexpected failures.

Ideally, the rolling update controller would take application readiness into account, and would ensure that a sufficient number of pods were productively serving at any given time.

The two replication controllers would need to create pods with at least one differentiating label, such as the image tag of the primary container of the pod, since it is typically image updates that motivate rolling updates.

Multiple release tracks

In addition to running multiple releases of an application while a rolling update is in progress, it's common to run multiple releases for an extended period of time, or even continuously, using multiple release tracks. The tracks would be differentiated by labels.

For instance, a service might target all pods with `tier in (frontend), environment in (prod)`. Now say you have 10 replicated pods that make up this tier. But you want to be able to 'canary' a new version of this component. You could set up a `replicationController` with `replicas` set to 9 for the bulk of the replicas, with labels `tier=frontend, environment=prod, track=stable`, and another `replicationController` with `replicas` set to 1 for the canary, with labels `tier=frontend, environment=prod, track=canary`. Now the service is covering both the canary and non-canary pods. But you can mess with the `replicationControllers` separately to test things out, monitor the results, etc.

Services in Kubernetes

Overview

Kubernetes Pods are ephemeral. They can come and go over time, especially when driven by things like ReplicationControllers. While each pod gets its own IP address, those IP addresses can not be relied upon to be stable over time. This leads to a problem: if some set of pods (let's call them backends) provides functionality to other pods (let's call them frontends) inside the Kubernetes cluster, how do those frontends find the backends?

Enter services.

A Kubernetes service is an abstraction which defines a logical set of pods and a policy by which to access them - sometimes called a micro-service. The goal of services is to provide a bridge for non-Kubernetes-native applications to access backends without the need to write code that is specific to Kubernetes. A service offers clients an IP and port pair which, when accessed, redirects to the appropriate backends. The set of pods targetted is determined by a label selector.

As an example, consider an image-process backend which is running with 3 live replicas. Those replicas are fungible - frontends do not care which backend they use. While the actual pods that comprise the set may change, the frontend client(s) do not need to know that. The service abstraction enables this decoupling.

Defining a service

A `service` in Kubernetes is a REST object, similar to a `pod`. Like a `pod`, a `service` definition can be POSTed to the apiserver to create a new instance. For example, suppose you have a set of `pods` that each expose port 9376 and carry a label "app=MyApp".

```
{
  "id": "myapp",
  "selector": {
    "app": "MyApp"
  },
  "containerPort": 9376,
  "protocol": "TCP",
  "port": 8765
}
```

This specification will create a new `service` named "myapp" which resolves to TCP port 9376 on any `pod` with the "app=MyApp" label. To access this `service`, a client can simply connect to $MYAPP_SERVICE_HOST on port $MYAPP_SERVICE_PORT.

How do they work?

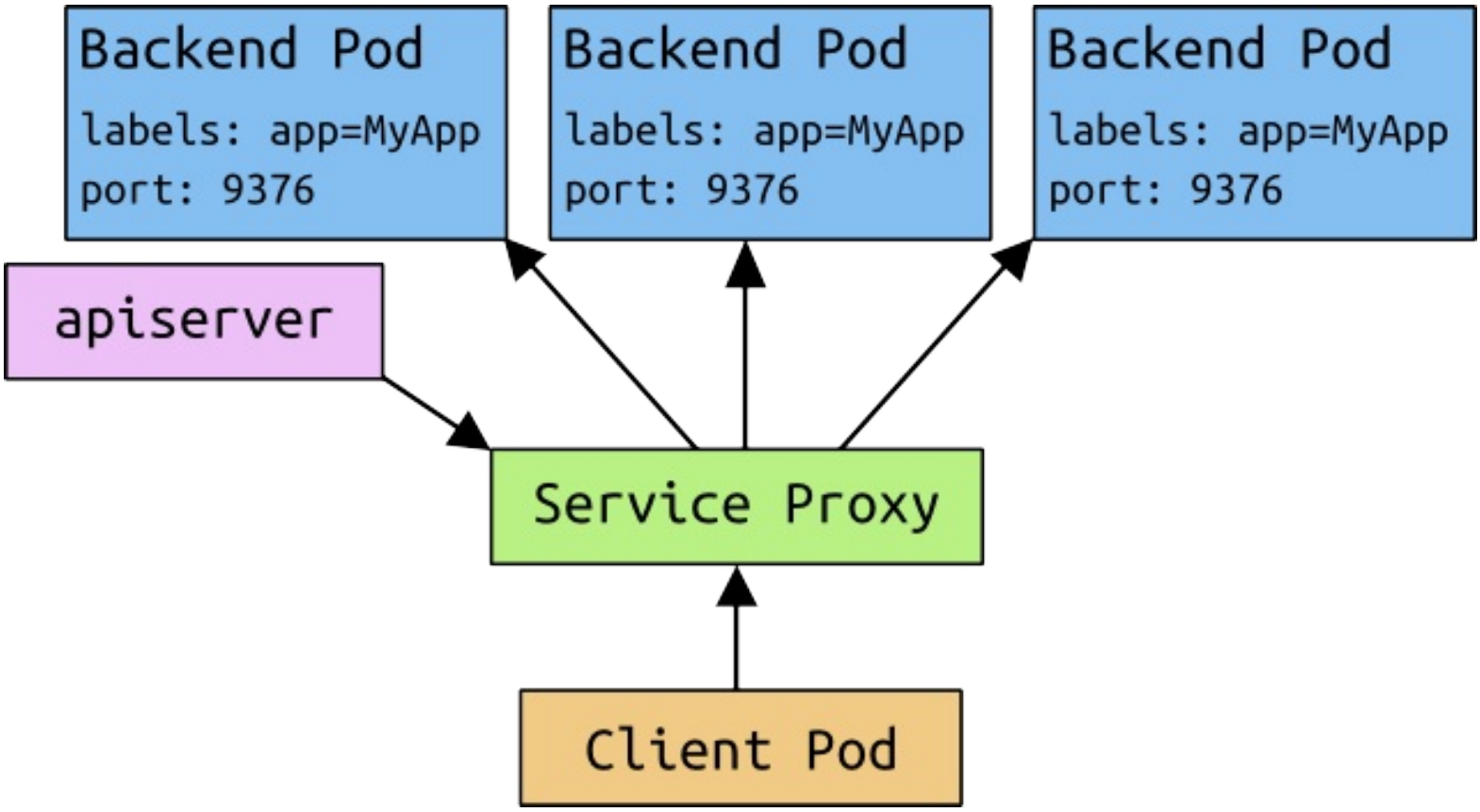
Each node in a Kubernetes cluster runs a `service proxy`. This application watches the Kubernetes master for the addition and removal of `service` objects and `endpoints` (pods that satisfy a service's label selector), and maintains a mapping of `service` to list of `endpoints`. It opens a port on the local node for each `service` and forwards traffic to backends (ostensibly according to a policy, but the only policy supported for now is round-robin).

When a `pod` is scheduled, the master adds a set of environment variables for each active `service`. We support both Docker-links-compatible variables (see makeLinkVariables) and simpler {SVCNAME}_SERVICE_HOST and {SVCNAME}_SERVICE_PORT variables, where the service name is upper-cased and dashes are converted to underscores. For example, the service "redis-master" exposed on TCP port 6379 and allocated IP address 10.0.0.11 produces the following environment variables:

```
REDIS_MASTER_SERVICE_HOST=10.0.0.11
REDIS_MASTER_SERVICE_PORT=6379
REDIS_MASTER_PORT=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP=tcp://10.0.0.11:6379
REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
REDIS_MASTER_PORT_6379_TCP_PORT=6379
REDIS_MASTER_PORT_6379_TCP_ADDR=10.0.0.11
```

This does imply an ordering requirement - any `service` that a `pod` wants to access must be created before the `pod` itself, or else the environment variables will not be populated. This restriction will be removed once DNS for `services` is supported.

A `service`, through its label selector, can resolve to 0 or more `endpoints`. Over the life of a `service`, the set of `pods` which comprise that `service` can grow, shrink, or turn over completely. Clients will only see issues if they are actively using a backend when that backend is removed from the `services` (and even then, open connections will persist for some protocols).



Services overview diagram

The gory details

The previous information should be sufficient for many people who just want to use `services`. However, there is a lot going on behind the scenes that may be worth understanding.

Avoiding collisions

One of the primary philosophies of Kubernetes is that users should not be exposed to situations that could cause their actions to fail through no fault of their own. In this situation, we are looking at network ports - users should not have to choose a port number if that choice might collide with another user. That is an isolation failure.

In order to allow users to choose a port number for their `services`, we must ensure that no two `services` can collide. We do that by allocating each `service` its own IP address.

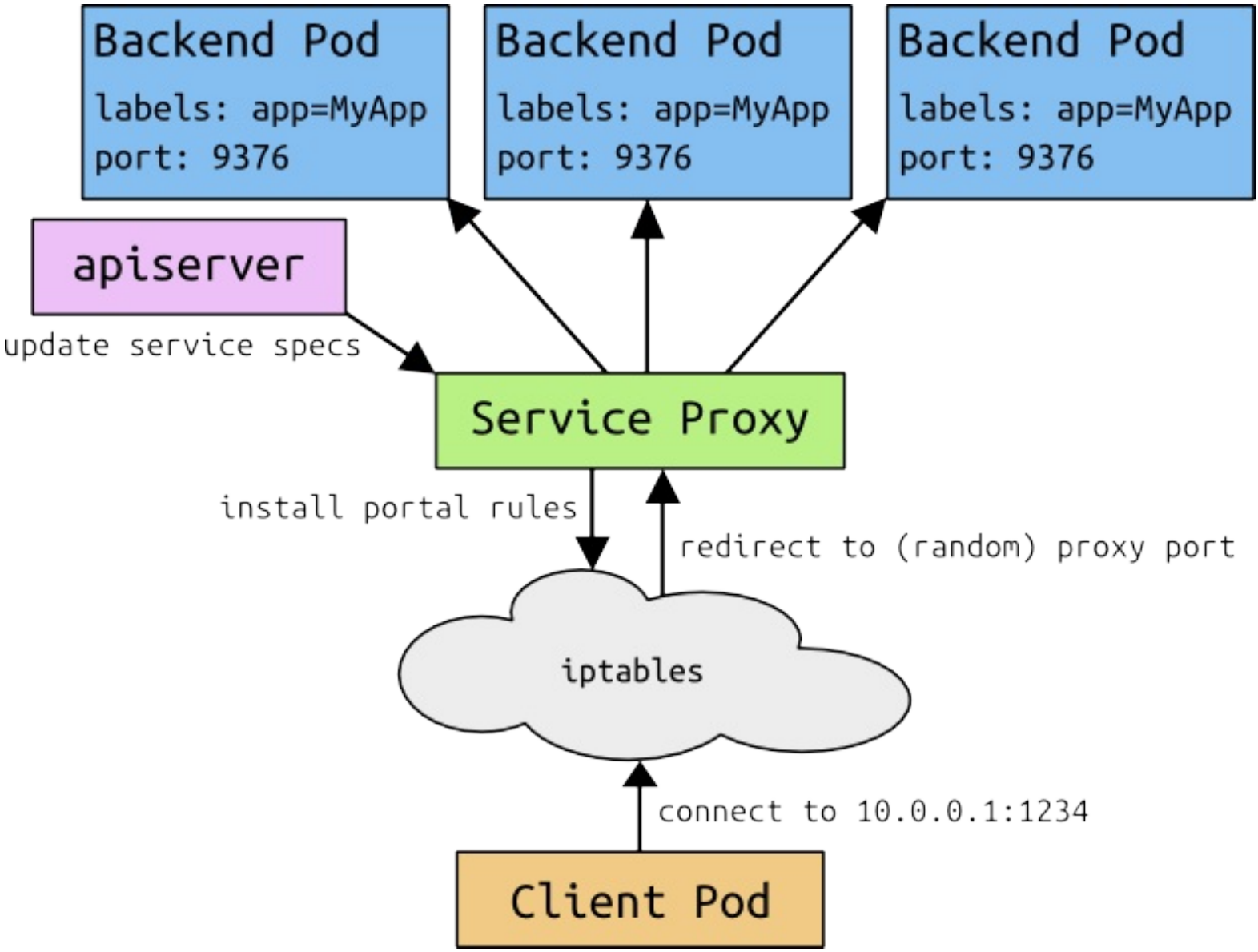
IPs and Portals

Unlike `pod` IP addresses, which actually route to a fixed destination, `service` IPs are not actually answered by a single host. Instead, we use `iptables` (packet processing logic in Linux) to define "virtual" IP addresses which are transparently redirected as needed. We call the tuple of the `service` IP and the `service` port the `portal`. When clients connect to the `portal`, their traffic is automatically transported to an appropriate endpoint. The environment variables for `services` are actually populated in terms of the portal IP and port. We will be adding DNS support for `services`, too.

As an example, consider the image processing application described above. When the backend `service` is created, the Kubernetes master assigns a portal IP address, for example 10.0.0.1. Assuming the `service` port is 1234, the portal is 10.0.0.1:1234. The master stores that information, which is then observed by all of the `service proxy` instances in the cluster. When a proxy sees a new portal, it opens a new random port, establishes an iptables redirect from the portal to this new port, and starts accepting connections on it.

When a client connects to `MYAPP_SERVICE_HOST` on the portal port (whether they know the port statically or look it up as MYAPP_SERVICE_PORT), the iptables rule kicks in, and redirects the packets to the `service proxy`'s own port. The `service proxy` chooses a backend, and starts proxying traffic from the client to the backend.

The net result is that users can choose any `service` port they want without risk of collision. Clients can simply connect to an IP and port, without being aware of which `pods` they are accessing.



Services detailed diagram

External Services

For some parts of your application (e.g. frontend) you want to expose a service on an external (publically visible) IP address.

If you want your service to be exposed on an external IP address, you can optionally supply a list of `publicIPs` which the `service` should respond to. These IP address will be combined with the `service`'s port and will also be mapped to the set of `pods` selected by the `service`. You are then responsible for ensuring that traffic to that external IP address gets sent to one or more Kubernetes worker nodes. An IPTables rules on each host that maps packets from the specified public IP address to the service proxy in the same manner as internal service IP addresses.

On cloud providers which support external load balancers, there is a simpler way to achieve the same thing. On such providers (e.g. GCE) you can leave `publicIPs` empty, and instead you can set the `createExternalLoadBalancer` flag on the service. This sets up a cloud-provider-specific load balancer (assuming that it is supported by your cloud provider) and populates the Public IP field with the appropriate value.

Shortcomings

We expect that using iptables for portals will work at small scale, but will not scale to large clusters with thousands of services. See the original design proposal for portals for more details.

Future work

In the future we envision that the proxy policy can become more nuanced than simple round robin balancing, for example master elected or sharded. We also envision that some `services` will have "real" load balancers, in which case the portal will simply transport the packets there.

Volumes

This document describes the current state of Volumes in kubernetes. Familiarity with pods is suggested.

A Volume is a directory, possibly with some data in it, which is accessible to a Container. Kubernetes Volumes are similar to but not the same as Docker Volumes.

A Pod specifies which Volumes its containers need in its ContainerManifest property.

A process in a Container sees a filesystem view composed from two sources: a single Docker image and zero or more Volumes. A Docker image is at the root of the file hierarchy. Any Volumes are mounted at points on the Docker image; Volumes do not mount on other Volumes and do not have hard links to other Volumes. Each container in the Pod independently specifies where on its image to mount each Volume. This is specified a VolumeMounts property.

Resources

The storage media (Disk, SSD, or memory) of a volume is determined by the media of the filesystem holding the kubelet root dir (typically `/var/lib/kubelet`). There is no limit on how much space an EmptyDir or PersistentDir volume can consume, and no isolation between containers or between pods.

In the future, we expect that a Volume will be able to request a certain amount of space using a resource specification, and to select the type of media to use, for clusters that have several media types.

Types of Volumes

Kubernetes currently supports three types of Volumes, but more may be added in the future.

EmptyDir

An EmptyDir volume is created when a Pod is bound to a Node. It is initially empty, when the first Container command starts. Containers in the same pod can all read and write the same files in the EmptyDir. When a Pod is unbound, the data in the EmptyDir is deleted forever.

Some uses for an EmptyDir are: - scratch space, such as for a disk-based mergesort or checkpointing a long computation. - a directory that a content-manager container fills with data while a webserver container serves the data.

Currently, the user cannot control what kind of media is used for an EmptyDir. If the Kubelet is configured to use a disk drive, then all EmptyDirectories will be created on that disk drive. In the future, it is expected that Pods can control whether the EmptyDir is on a disk drive, SSD, or tmpfs.

HostDir

A Volume with a HostDir property allows access to files on the current node.

Some uses for a HostDir are: - running a container that needs access to Docker internals; use a HostDir of /var/lib/docker. - running cAdvisor in a container; use a HostDir of /dev/cgroups.

Watch out when using this type of volume, because: - pods with identical configuration (such as created from a podTemplate) may behave differently on different nodes due to different files on different nodes. - When Kubernetes adds resource-aware scheduling, as is planned, it will not be able to account for resources used by a HostDir.

GCEPersistentDisk

Important: You must create a PD using `gcloud` or the GCE API before you can use it

A Volume with a GCEPersistentDisk property allows access to files on a Google Compute Engine (GCE) Persistent Disk.

There are some restrictions when using a GCEPersistentDisk: - the nodes (what the kubelet runs on) need to be GCE VMs - those VMs need to be in the same GCE project and zone as the PD - avoid creating multiple pods that use the same Volume if any mount it read/write. - if a pod P already mounts a volume read/write, and a second pod Q attempts to use the volume, regardless of if it tries to use it read-only or read/write, Q will fail. - if a pod P already mounts a volume read-only, and a second pod Q attempts to use the volume read/write, Q will fail. - replication controllers with replicas > 1 can only be created for pods that use read-only mounts.

Creating a PD

Before you can use a GCE PD with a pod, you need to create it.

```
gcloud compute disks create --size=500GB --zone=us-central1-a my-data-disk
```

GCE PD Example configuration:

```
apiVersion: v1beta1
desiredState:
  manifest:
    containers:
      - image: kubernetes/pause
        name: testpd
```

```
        volumeMounts:
          - mountPath: "/testpd"
            name: "testpd"
    id: testpd
    version: v1beta1
    volumes:
      - name: testpd
        source:
          persistentDisk:
            # This GCE PD must already exist.
            pdName: test
            fsType: ext4
id: testpd
kind: Pod
```

Note that the model described in this document has not yet been implemented. The tracking issue for implementation of this model is #168. Currently, only memory and cpu limits on containers (not pods) are supported. "memory" is in bytes and "cpu" is in milli-cores.

The Kubernetes resource model

To do good pod placement, Kubernetes needs to know how big pods are, as well as the sizes of the nodes onto which they are being placed. The definition of "how big" is given by the Kubernetes resource model - the subject of this document.

The resource model aims to be: * simple, for common cases; * extensible, to accommodate future growth; * regular, with few special cases; and * precise, to avoid misunderstandings and promote pod portability.

The resource model

A Kubernetes *resource* is something that can be requested by, allocated to, or consumed by a pod or container. Examples include memory (RAM), CPU, disk-time, and network bandwidth.

Once resources on a node have been allocated to one pod, they should not be allocated to another until that pod is removed or exits. This means that Kubernetes schedulers should ensure that the sum of the resources allocated (requested and granted) to its pods never exceeds the usable capacity of the node. Testing whether a pod will fit on a node is called *feasibility checking*.

Note that the resource model currently prohibits over-committing resources; we will want to relax that restriction later.

Resource types

All resources have a *type* that is identified by their *typename* (a string, e.g., "memory"). Several resource types are predefined by Kubernetes (a full list is below), although only two will be supported at first: CPU and memory. Users and system administrators can define their own resource types if they wish (e.g., Hadoop slots).

A fully-qualified resource typename is constructed from a DNS-style *subdomain*, followed by a slash `/`, followed by a DNS Label. * The subdomain must conform to RFC 1035 / RFC 1123 (e.g., `kubernetes.io`, `myveryown.org`). * The DNS label must conform to RFC 1123 - alphanumeric characters, with a maximum length of 63 characters, with the '-' character allowed anywhere except the first or last character. * As a shorthand, any resource typename that does not start with a subdomain and a slash will automatically be prefixed with the built-in Kubernetes *namespace*, `kubernetes.io/` in order to fully-qualify it. This namespace is reserved for code in the open source Kubernetes repository; as a result, all user typenames MUST be fully qualified, and cannot be created in this namespace.

Some example typenames include `memory` (which will be fully-qualified as `kubernetes.io/memory`), and `myveryown.org/shiny-new-resource`.

For future reference, note that some resources, such as CPU and network bandwidth, are *compressible*, which means that their usage can potentially be throttled in a relatively benign manner. All other resources are *incompressible*, which means that any attempt to throttle them is likely to cause grief. This distinction will be important if a Kubernetes implementation supports over-committing of resources.

Resource quantities

Initially, all Kubernetes resource types are *quantitative*, and have an associated *unit* for quantities of the associated resource (e.g., bytes for memory, bytes per seconds for bandwidth, instances for software licences). The units will always be a resource type's natural base units (e.g., bytes, not MB), to avoid confusion between binary and decimal multipliers and the underlying unit multiplier (e.g., is memory measured in MiB, MB, or GB?).

Resource quantities can be added and subtracted: for example, a node has a fixed quantity of each resource type that can be allocated to pods/containers; once such an allocation has been made, the allocated resources cannot be made available to other pods/containers without over-committing the resources.

To make life easier for people, quantities can be represented externally as unadorned integers, or as fixed-point integers with one of these SI suffices (E, P, T, G, M, K, m) or their power-of-two equivalents (Ei, Pi, Ti, Gi, Mi, Ki). For example, the following represent roughly the same value: 128974848, "129e6", "129M" , "123Mi". Small quantities can be represented directly as decimals (e.g., 0.3), or using milli-units (e.g., "300m"). * "Externally" means in user interfaces, reports, graphs, and in JSON or YAML resource specifications that might be generated or read by people. * Case is significant: "m" and "M" are not the same, so "k" is not a valid SI suffix. There are no power-of-two equivalents for SI suffixes that represent multipliers less than 1. * These conventions only apply to resource quantities, not arbitrary values.

Internally (i.e., everywhere else), Kubernetes will represent resource quantities as integers so it can avoid

problems with rounding errors, and will not use strings to represent numeric values. To achieve this, quantities that naturally have fractional parts (e.g., CPU seconds/second) will be scaled to integral numbers of milli-units (e.g., milli-CPUs) as soon as they are read in. Internal APIs, data structures, and protobufs will use these scaled integer units. Raw measurement data such as usage may still need to be tracked and calculated using floating point values, but internally they should be rescaled to avoid some values being in milli-units and some not. * Note that reading in a resource quantity and writing it out again may change the way its values are represented, and truncate precision (e.g., 1.0001 may become 1.000), so comparison and difference operations (e.g., by an updater) must be done on the internal representations. * Avoiding milli-units in external representations has advantages for people who will use Kubernetes, but runs the risk of developers forgetting to rescale or accidentally using floating-point representations. That seems like the right choice. We will try to reduce the risk by providing libraries that automatically do the quantization for JSON/YAML inputs.

Resource specifications

Both users and a number of system components, such as schedulers, (horizontal) auto-scalers, (vertical) auto-sizers, load balancers, and worker-pool managers need to reason about resource requirements of workloads, resource capacities of nodes, and resource usage. Kubernetes divides specifications of *desired state*, aka the Spec, and representations of *current state*, aka the Status. Resource requirements and total node capacity fall into the specification category, while resource usage, characterizations derived from usage (e.g., maximum usage, histograms), and other resource demand signals (e.g., CPU load) clearly fall into the status category and are discussed in the Appendix for now.

Resource requirements for a container or pod should have the following form:

```
resourceRequirementSpec: [
  request:   [ cpu: 2.5, memory: "40Mi" ],
  limit:     [ cpu: 4.0, memory: "99Mi" ],
]
```

Where: * *request* [optional]: the amount of resources being requested, or that were requested and have been allocated. Scheduler algorithms will use these quantities to test feasibility (whether a pod will fit onto a node). If a container (or pod) tries to use more resources than its *request*, any associated SLOs are voided - e.g., the program it is running may be throttled (compressible resource types), or the attempt may be denied. If *request* is omitted for a container, it defaults to *limit* if that is explicitly specified, otherwise to an implementation-defined value; this will always be 0 for a user-defined resource type. If *request* is omitted for a pod, it defaults to the sum of the (explicit ior implicit) *request* values for the containers it encloses.

- *limit* [optional]: an upper bound or cap on the maximum amount of resources that will be made available to a container or pod; if a container or pod uses more resources than its *limit*, it may be terminated. The *limit* defaults to "unbounded"; in practice, this probably means the capacity of an enclosing container, pod, or node, but may result in non-deterministic behavior, especially for memory.

Total capacity for a node should have a similar structure:

```
resourceCapacitySpec: [
  total:     [ cpu: 12,  memory: "128Gi" ]
]
```

Where: * *total*: the total allocatable resources of a node. Initially, the resources at a given scope will bound the resources of the sum of inner scopes.

Notes

- It is an error to specify the same resource type more than once in each list.
- It is an error for the *request* or *limit* values for a pod to be less than the sum of the (explicit or defaulted) values for the containers it encloses. (We may relax this later.)

- If multiple pods are running on the same node and attempting to use more resources than they have requested, the result is implementation-defined. For example: unallocated or unused resources might be spread equally across claimants, or the assignment might be weighted by the size of the original request, or as a function of limits, or priority, or the phase of the moon, perhaps modulated by the direction of the tide. Thus, although it's not mandatory to provide a *request*, it's probably a good idea. (Note that the *request* could be filled in by an automated system that is observing actual usage and/or historical data.)

- Internally, the Kubernetes master can decide the defaulting behavior and the kubelet implementation may expected an absolute specification. For example, if the master decided that "the default is unbounded" it would pass 2^64 to the kubelet.

Kubernetes-defined resource types

The following resource types are predefined ("reserved") by Kubernetes in the `kubernetes.io` namespace, and so cannot be used for user-defined resources. Note that the syntax of all resource types in the resource spec is deliberately similar, but some resource types (e.g., CPU) may receive significantly more support than simply tracking quantities in the schedulers and/or the Kubelet.

Processor cycles

- Name: `cpu` (or `kubernetes.io/cpu`)
- Units: Kubernetes Compute Unit seconds/second (i.e., CPU cores normalized to a canonical "Kubernetes CPU")
- Internal representation: milli-KCUs
- Compressible? yes
- Qualities: this is a placeholder for the kind of thing that may be supported in the future -- see #147
    - [future] `schedulingLatency`: as per lmctfy
    - [future] `cpuConversionFactor`: property of a node: the speed of a CPU core on the node's processor divided by the speed of the canonical Kubernetes CPU (a floating point value; default = 1.0).

To reduce performance portability problems for pods, and to avoid worse-case provisioning behavior, the units of CPU will be normalized to a canonical "Kubernetes Compute Unit" (KCU, pronounced ˈko͝okōo͞), which will roughly be equivalent to a single CPU hyperthreaded core for some recent x86 processor. The normalization may be implementation-defined, although some reasonable defaults will be provided in the open-source Kubernetes code.

Note that requesting 2 KCU won't guarantee that precisely 2 physical cores will be allocated - control of aspects like this will be handled by resource *qualities* (a future feature).

Memory

- Name: `memory` (or `kubernetes.io/memory`)
- Units: bytes
- Compressible? no (at least initially)

The precise meaning of what "memory" means is implementation dependent, but the basic idea is to rely on the underlying `memcg` mechanisms, support, and definitions.

Note that most people will want to use power-of-two suffixes (Mi, Gi) for memory quantities rather than decimal ones: "64MiB" rather than "64MB".

Resource metadata

A resource type may have an associated read-only ResourceType structure, that contains metadata about the type. For example:

```
resourceTypes: [
  "kubernetes.io/memory": [
    isCompressible: false, ...
  ]
  "kubernetes.io/cpu": [
    isCompressible: true, internalScaleExponent: 3, ...
  ]
  "kubernetes.io/disk-space": [ ... }
]
```

Kubernetes will provide ResourceType metadata for its predefined types. If no resource metadata can be found for a resource type, Kubernetes will assume that it is a quantified, incompressible resource that is not specified in milli-units, and has no default value.

The defined properties are as follows:

field name	type	contents
name	string, required	the typename, as a fully-qualified string (e.g., `kubernetes.io/cpu`)
internalScaleExponent	int, default=0	external values are multiplied by 10 to this power for internal storage (e.g., 3 for milli-units)
units	string, required	format: `unit* [per unit+]` (e.g., `second`, `byte per second`). An empty unit field means "dimensionless".
isCompressible	bool, default=false	true if the resource type is compressible
defaultRequest	string, default=none	in the same format as a user-supplied value
[future] quantization	number, default=1	smallest granularity of allocation: requests may be rounded up to a multiple of this unit; implementation-defined unit (e.g., the page size for RAM).

Appendix: future extensions

The following are planned future extensions to the resource model, included here to encourage comments.

Usage data

Because resource usage and related metrics change continuously, need to be tracked over time (i.e., historically), can be characterized in a variety of ways, and are fairly voluminous, we will not include usage in core API objects, such as Pods and Nodes, but will provide separate APIs for accessing and managing that data. See the Appendix for possible representations of usage data, but the representation we'll use is TBD.

Singleton values for observed and predicted future usage will rapidly prove inadequate, so we will support the following structure for extended usage information:

```
resourceStatus: [
  usage:     [ cpu: <CPU-info>, memory: <memory-info> ],
  maxusage:  [ cpu: <CPU-info>, memory: <memory-info> ],
  predicted: [ cpu: <CPU-info>, memory: <memory-info> ],
]
```

where a `<CPU-info>` or `<memory-info>` structure looks like this:

```
{
    mean: <value>    # arithmetic mean
    max: <value>     # minimum value
    min: <value>     # maximum value
    count: <value>   # number of data points
    percentiles: [   # map from %iles to values
      "10": <10th-percentile-value>,
      "50": <median-value>,
      "99": <99th-percentile-value>,
      "99.9": <99.9th-percentile-value>,
      ...
    ]
 }
```

All parts of this structure are optional, although we strongly encourage including quantities for 50, 90, 95, 99, 99.5, and 99.9 percentiles. *[In practice, it will be important to include additional info such as the length of the time window over which the averages are calculated, the confidence level, and information-quality metrics such as the number of dropped or discarded data points.]* and predicted

Future resource types

[future] Network bandwidth

- Name: "network-bandwidth" (or `kubernetes.io/network-bandwidth`)
- Units: bytes per second
- Compressible? yes

[future] Network operations

- Name: "network-iops" (or `kubernetes.io/network-iops`)
- Units: operations (messages) per second
- Compressible? yes

[future] Storage space

- Name: "storage-space" (or `kubernetes.io/storage-space`)
- Units: bytes
- Compressible? no

The amount of secondary storage space available to a container. The main target is local disk drives and SSDs, although this could also be used to qualify remotely-mounted volumes. Specifying whether a resource is a raw disk, an SSD, a disk array, or a file system fronting any of these, is left for future work.

[future] Storage time

- Name: storage-time (or `kubernetes.io/storage-time`)
- Units: seconds per second of disk time
- Internal representation: milli-units
- Compressible? yes

This is the amount of time a container spends accessing disk, including actuator and transfer time. A standard disk drive provides 1.0 diskTime seconds per second.

[future] Storage operations

- Name: "storage-iops" (or `kubernetes.io/storage-iops`)
- Units: operations per second
- Compressible? yes

Kubernetes UI instructions

Kubernetes User Interface

Kubernetes currently supports a simple web user interface.

Running locally

Start the server:

```
cluster/kubectl.sh proxy -www=$PWD/www
```

The UI should now be running on localhost

Running remotely

When Kubernetes is deployed, the server deploys the UI, you can visit `/static/index.html#/groups//selector` on your master server.

Interacting with the user interface.

The Kubernetes user interface is a query-based visualization of the Kubernetes API. The user interface is defined by two functional primitives:

GroupBy

GroupBy takes a label `key` as a parameter, places all objects with the same value for that key within a single group. For example `/groups/host/selector` groups pods by host. `/groups/name/selector` groups pods by name. Groups are hiearchical, for example `/groups/name/host/selector` first groups by pod name, and then by host.

Select

Select takes a label selector and uses it to filter, so only resources which match that label selector are displayed. For example, `/groups/host/selector/name=frontend`, shows pods, grouped by host, which have a label with the name `frontend`.

Rebuilding the UI

The UI relies on go-bindata

To install go-bindata:

```
go get github.com/jteeuwen/go-bindata/...
```

To rebuild the UI, run the following:

```
hack/build-ui.sh
```

Kubernetes Roadmap

Updated Feb 9, 2015

This document is intended to capture the set of supported use cases, features, docs, and patterns that we feel are required to call Kubernetes “feature complete” for a 1.0 release candidate. This list does not emphasize the bug fixes and stabilization that will be required to take it all the way to production ready. This is a living document, and is certainly open for discussion.

Target workloads

Most realistic examples of production services include a load-balanced web frontend exposed to the public Internet, with a stateful backend, such as a clustered database or key-value store. We will target such workloads for our 1.0 release.

APIs and core features

1. Consistent v1 API

- Status: v1beta3 (#1519) is being developed as the release candidate for the v1 API.

2. Multi-port services for apps which need more than one port on the same portal IP (#1802)

- Status: #2585 covers the design.

3. Nominal services for applications which need one stable IP per pod instance (#260)

- Status: #2585 covers some design options.

4. API input is scrubbed of status fields in favor of a new API to set status (#4248)

- Status: in progress

5. Input validation reporting versioned field names (#2518)

- Status: in progress

6. Error reporting: Report common problems in ways that users can discover

- Status:

7. Event management: Make events usable and useful

- Status:

8. Persistent storage support (#4055)

- Status: in progress

9. Allow nodes to join/leave a cluster (#2303,#2435)

- Status: high level design doc.

10. Handle node death

- Status: mostly covered by nodes joining/leaving a cluster

11. Allow live cluster upgrades (#2524)

- Status: design in progress

12. Allow kernel upgrades

- Status: mostly covered by nodes joining/leaving a cluster, need demonstration

13. Allow rolling-updates to fail gracefully (#1353)

- Status:

14. Easy .dockercfg

- Status:

15. Demonstrate cluster stability over time

- Status

16. Kubelet use the kubernetes API to fetch jobs to run (instead of etcd) on supported platforms

- Status

Reliability and performance

1. Restart system components in case of crash (#2884)

- Status: in progress

2. Scale to 100 nodes (#3876)

- Status: in progress

3. Scale to 30-50 pods (1-2 containers each) per node (#4188)

- Status:

4. Scheduling throughput: 99% of scheduling decisions made in less than 1s on 100 node, 3000 pod cluster; linear time to number of nodes and pods (#3954)

- Status:

5. API performance: 99% of API calls return in less than 1s; constant time to number of nodes and pods (#4521)

- Status:

6. Manage and report disk space on nodes (#4135)

- Status: in progress

7. API test coverage more than 85% in e2e tests

- Status:

Project

1. Define a deprecation policy for expiring and removing features and interfaces, including the time non-beta APIs will be supported

- Status:

2. Define a version numbering policy regarding point upgrades, support, compat, and release frequency.

- Status:

3. Define an SLO that users can reasonable expect to hit in properly managed clusters

- Status:

4. Accurate and complete API documentation

- Status:

5. Accurate and complete getting-started-guides for supported platforms

- Status:

Platforms

1. Possible for cloud partners / vendors to self-qualify Kubernetes on their platform.

- Status:

2. Define the set of platforms that are supported by the core team.

- Status:

Beyond 1.0

We acknowledge that there are a great many things that are not included in our 1.0 roadmap. We intend to document the plans past 1.0 soon, but some of the things that are clearly in scope include:

1. Scalability - more nodes, more pods
2. HA masters
3. Monitoring
4. Authn and authz
5. Enhanced resource management and isolation
6. Better performance
7. Easier plugins and add-ons
8. More support for jobs that complete (compute, batch)
9. More platforms
10. Easier testing

Using Salt to configure Kubernetes

The Kubernetes cluster can be configured using Salt.

The Salt scripts are shared across multiple hosting providers, so it's important to understand some background information prior to making a modification to ensure your changes do not break hosting Kubernetes across multiple environments. Depending on where you host your Kubernetes cluster, you may be using different operating systems and different networking configurations. As a result, it's important to understand some background information before making Salt changes in order to minimize introducing failures for other hosting providers.

Salt cluster setup

The **salt-master** service runs on the kubernetes-master node.

The **salt-minion** service runs on the kubernetes-master node and each kubernetes-minion node in the cluster.

Each salt-minion service is configured to interact with the **salt-master** service hosted on the kubernetes-master via the **master.conf** file.

```
[root@kubernetes-master] $ cat /etc/salt/minion.d/master.conf
master: kubernetes-master
```

The salt-master is contacted by each salt-minion and depending upon the machine information presented, the salt-master will provision the machine as either a kubernetes-master or kubernetes-minion with all the required capabilities needed to run Kubernetes.

If you are running the Vagrant based environment, the **salt-api** service is running on the kubernetes-master. It is configured to enable the vagrant user to introspect the salt cluster in order to find out about machines in the Vagrant environment via a REST API.

Salt security

Security is not enabled on the salt-master, and the salt-master is configured to auto-accept incoming requests from minions. It is not recommended to use this security configuration in production environments without deeper study. (In some environments this isn't as bad as it might sound if the salt master port isn't externally accessible and you trust everyone on your network.)

```
[root@kubernetes-master] $ cat /etc/salt/master.d/auto-accept.conf
open_mode: True
auto_accept: True
```

Salt minion configuration

Each minion in the salt cluster has an associated configuration that instructs the salt-master how to provision the required resources on the machine.

An example file is presented below using the Vagrant based environment.

```
[root@kubernetes-master] $ cat /etc/salt/minion.d/grains.conf
grains:
  master_ip: $MASTER_IP
  etcd_servers: $MASTER_IP
  cloud_provider: vagrant
  roles:
    - kubernetes-master
```

Each hosting environment has a slightly different grains.conf file that is used to build conditional logic where required in the Salt files.

The following enumerates the set of defined key/value pairs that are supported today. If you add new ones, please make sure to update this list.

Key	Value
`apiservers`	(Optional) The IP address / host name where a kubelet can get read-only access to kube-apiserver
`cbr-cidr`	(Optional) The minion IP address range used for the docker container bridge.
`cloud`	(Optional) Which IaaS platform is used to host kubernetes, *gce*, *azure*, *aws*, *vagrant*
`cloud_provider`	(Optional) The cloud_provider used by apiserver: *gce*, *azure*, *vagrant*
`etcd_servers`	(Optional) Comma-delimited list of IP addresses the kube-apiserver and kubelet use to reach etcd. Uses the IP of the first machine in the kubernetes_master role.
`hostnamef`	(Optional) The full host name of the machine, i.e. hostname -f
`master_ip`	(Optional) The IP address that the kube-apiserver will bind against
`node_ip`	(Optional) The IP address to use to address this node
`minion_ip`	(Optional) Mapped to the kubelet hostname_override, K8S TODO - change this name
`network_mode`	(Optional) Networking model to use among nodes: *openvswitch*
`networkInterfaceName`	(Optional) Networking interface to use to bind addresses, default value *eth0*
`publicAddressOverride`	(Optional) The IP address the kube-apiserver should use to bind against for external read-only access
`roles`	(Required) 1. `kubernetes-master` means this machine is the master in the kubernetes cluster. 2. `kubernetes-pool` means this machine is a kubernetes-minion. Depending on the role, the Salt scripts will provision different resources on the machine.

These keys may be leveraged by the Salt sls files to branch behavior.

In addition, a cluster may be running a Debian based operating system or Red Hat based operating system (Centos, Fedora, RHEL, etc.). As a result, its important to sometimes distinguish behavior based on operating system using if branches like the following.

```
{% if grains['os_family'] == 'RedHat' %}
// something specific to a RedHat environment (Centos, Fedora, RHEL) where you may
{% else %}
// something specific to Debian environment (apt-get, initd)
```

{% endif %}

Best Practices

1. When configuring default arguments for processes, its best to avoid the use of EnvironmentFiles (Systemd in Red Hat environments) or init.d files (Debian distributions) to hold default values that should be common across operating system environments. This helps keep our Salt template files easy to understand for editors that may not be familiar with the particulars of each distribution.

Future enhancements (Networking)

Per pod IP configuration is provider specific, so when making networking changes, its important to sand-box these as all providers may not use the same mechanisms (iptables, openvswitch, etc.)

We should define a grains.conf key that captures more specifically what network configuration environment is being used to avoid future confusion across providers.

Cloud Native Deployments of Cassandra using Kubernetes

The following document describes the development of a *cloud native* Cassandra deployment on Kubernetes. When we say *cloud native* we mean an application which understands that it is running within a cluster manager, and uses this cluster management infrastructure to help implement the application. In particular, in this instance, a custom Cassandra `SeedProvider` is used to enable Cassandra to dynamically discover new Cassandra nodes as they join the cluster.

This document also attempts to describe the core components of Kubernetes, *Pods*, *Services* and *Replication Controllers*.

Prerequisites

This example assumes that you have a Kubernetes cluster installed and running, and that you have installed the `kubectl` command line tool somewhere in your path. Please see the getting started for installation instructions for your platform.

A note for the impatient

This is a somewhat long tutorial. If you want to jump straight to the "do it now" commands, please see the tl;dr at the end.

Simple Single Pod Cassandra Node

In Kubernetes, the atomic unit of an application is a *Pod*. A Pod is one or more containers that *must* be scheduled onto the same host. All containers in a pod share a network namespace, and may optionally share mounted volumes. In this simple case, we define a single container running Cassandra for our pod:

```
id: cassandra
kind: Pod
apiVersion: v1beta1
desiredState:
  manifest:
    version: v1beta1
    id: cassandra
    containers:
      - name: cassandra
        image: kubernetes/cassandra
        command:
          - /run.sh
        cpu: 1000
        ports:
          - name: cql
            containerPort: 9042
          - name: thrift
            containerPort: 9160
        env:
          - key: MAX_HEAP_SIZE
            value: 512M
          - key: HEAP_NEWSIZE
            value: 100M
labels:
  name: cassandra
```

There are a few things to note in this description. First is that we are running the `kubernetes/cassandra` image. This is a standard Cassandra installation on top of Debian. However it also adds a custom SeedProvider to Cassandra. In Cassandra, a `SeedProvider` bootstraps the gossip protocol that Cassandra uses to find other nodes. The `KubernetesSeedProvider` discovers the Kubernetes API Server using the built in Kubernetes discovery service, and then uses the Kubernetes API to find new nodes (more

on this later)

You may also note that we are setting some Cassandra parameters (`MAX_HEAP_SIZE` and `HEAP_NEWSIZE`). We also tell Kubernetes that the container exposes both the `CQL` and `Thrift` API ports. Finally, we tell the cluster manager that we need 1000 milli-cpus (1 core).

Given this configuration, we can create the pod as follows

```
$ kubectl create -f cassandra.yaml
```

After a few moments, you should be able to see the pod running:

```
$ kubectl get pods cassandra
POD                 CONTAINER(S)        IMAGE(S)                HOST
cassandra           cassandra           kubernetes/cassandra    kubernetes-minion-1
```

Adding a Cassandra Service

In Kubernetes a *Service* describes a set of Pods that perform the same task. For example, the set of nodes in a Cassandra cluster, or even the single node we created above. An important use for a Service is to create a load balancer which distributes traffic across members of the set. But a *Service* can also be used as a standing query which makes a dynamically changing set of Pods (or the single Pod we've already created) available via the Kubernetes API. This is the way that we use initially use Services with Cassandra.

Here is the service description:

```
id: cassandra
kind: Service
apiVersion: v1beta1
port: 9042
containerPort: 9042
selector:
  name: cassandra
```

The important thing to note here is the `selector`. It is a query over labels, that identifies the set of *Pods* contained by the *Service*. In this case the selector is `name=cassandra`. If you look back at the Pod specification above, you'll see that the pod has the corresponding label, so it will be selected for membership in this Service.

Create this service as follows:

```
$ kubectl create -f cassandra-service.yaml
```

Once the service is created, you can query it's endpoints:

```
$ kubectl get endpoints cassandra -o yaml
apiVersion: v1beta1
creationTimestamp: 2015-01-05T05:51:50Z
endpoints:
- 10.244.1.10:9042
id: cassandra
kind: Endpoints
namespace: default
resourceVersion: 69130
selfLink: /api/v1beta1/endpoints/cassandra?namespace=default
uid: f1937b47-949e-11e4-8a8b-42010af0e23e
```

You can see that the *Service* has found the pod we created in step one.

Adding replicated nodes

Of course, a single node cluster isn't particularly interesting. The real power of Kubernetes and Cassandra lies in easily building a replicated, resizable Cassandra cluster.

In Kubernetes a *Replication Controller* is responsible for replicating sets of identical pods. Like a *Service* it has a selector query which identifies the members of it's set. Unlike a *Service* it also has a desired number of replicas, and it will create or delete *Pods* to ensure that the number of *Pods* matches up with it's desired state.

Replication Controllers will "adopt" existing pods that match their selector query, so let's create a Replication Controller with a single replica to adopt our existing Cassandra Pod.

```
id: cassandra
kind: ReplicationController
apiVersion: v1beta1
desiredState:
  replicas: 1
  replicaSelector:
    name: cassandra
  # This is identical to the pod config above
  podTemplate:
    desiredState:
      manifest:
        version: v1beta1
        id: cassandra
        containers:
          - name: cassandra
            image: kubernetes/cassandra
            command:
              - /run.sh
            cpu: 1000
            ports:
              - name: cql
                containerPort: 9042
              - name: thrift
                containerPort: 9160
            env:
              - key: MAX_HEAP_SIZE
                value: 512M
              - key: HEAP_NEWSIZE
                value: 100M
    labels:
      name: cassandra
```

The bulk of the replication controller config is actually identical to the Cassandra pod declaration above, it simply gives the controller a recipe to use when creating new pods. The other parts are the `replicaSelector` which contains the controller's selector query, and the `replicas` parameter which specifies the desired number of replicas, in this case 1.

Create this controller:

```
$ kubectl create -f cassandra-controller.yaml
```

Now this is actually not that interesting, since we haven't actually done anything new. Now it will get interesting.

Let's resize our cluster to 2:

```
$ kubectl resize rc cassandra --replicas=2
```

Now if you list the pods in your cluster, you should see two cassandra pods:

```
$ kubectl get pods
POD                                    CONTAINER(S)        IMAGE(S)
cassandra                              cassandra           kubernetes/cassandra
16b2beab-94a1-11e4-8a8b-42010af0e23e   cassandra           kubernetes/cassandra
```

Notice that one of the pods has the human readable name `cassandra` that you specified in your config before, and one has a random string, since it was named by the replication controller.

To prove that this all works, you can use the `nodetool` command to examine the status of the cluster, for example:

```
$ ssh 1.2.3.4
$ docker exec <cassandra-container-id> nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address      Load       Tokens  Owns (effective)  Host ID
UN  10.244.3.29  72.07 KB   256     100.0%            f736f0b5-bd1f-46f1-9b9d-7e8
UN  10.244.1.10  41.14 KB   256     100.0%            42617acd-b16e-4ee3-9486-68a
```

Now let's resize our cluster to 4 nodes:

```
$ kubectl resize rc cassandra --replicas=4
```

Examining the status again:

```
$ docker exec <cassandra-container-id> nodetool status
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address      Load       Tokens  Owns (effective)  Host ID
UN  10.244.3.29  72.07 KB   256     49.5%             f736f0b5-bd1f-46f1-9b9d-7e8
UN  10.244.2.14  61.62 KB   256     52.6%             3e9981a6-6919-42c4-b2b8-af5
UN  10.244.1.10  41.14 KB   256     49.5%             42617acd-b16e-4ee3-9486-68a
UN  10.244.4.8   63.83 KB   256     48.3%             eeb73967-d1e6-43c1-bb54-512
```

tl; dr;

For those of you who are impatient, here is the summary of the commands we ran in this tutorial.

```
# create a single cassandra node
kubectl create -f cassandra.yaml

# create a service to track all cassandra nodes
kubectl create -f cassandra-service.yaml

# create a replication controller to replicate cassandra nodes
kubectl create -f cassandra-controller.yaml

# scale up to 2 nodes
kubectl resize rc cassandra --replicas=2

# validate the cluster
docker exec <container-id> nodetool status

# scale up to 4 nodes
kubectl resize rc cassandra --replicas=4
```

Seed Provider Source

```
package io.k8s.cassandra;

import java.io.IOException;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.net.URL;
import java.net.URLConnection;
import java.security.KeyManagementException;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.annotate.JsonIgnoreProperties;
import org.codehaus.jackson.map.ObjectMapper;
import org.apache.cassandra.locator.SeedProvider;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class KubernetesSeedProvider implements SeedProvider {
    @JsonIgnoreProperties(ignoreUnknown = true)
    static class Endpoints {
        public String[] endpoints;
    }

    private static String getEnvOrDefault(String var, String def) {
        String val = System.getenv(var);
        if (val == null) {
        val = def;
        }
        return val;
    }

    private static final Logger logger = LoggerFactory.getLogger(KubernetesSeedPro

    private List defaultSeeds;

    public KubernetesSeedProvider(Map<String, String> params) {
        // Taken from SimpleSeedProvider.java
        // These are used as a fallback, if we get nothing from k8s.
        String[] hosts = params.get("seeds").split(",", -1);
        defaultSeeds = new ArrayList<InetAddress>(hosts.length);
        for (String host : hosts)
        {
        try {
            defaultSeeds.add(InetAddress.getByName(host.trim()));
        }
        catch (UnknownHostException ex)
            {
            // not fatal... DD will bark if there end up being zero seeds.
            logger.warn("Seed provider couldn't lookup host " + host);
            }
        }
    }
```

```
    public List<InetAddress> getSeeds() {
        List<InetAddress> list = new ArrayList<InetAddress>();
        String protocol = getEnvOrDefault("KUBERNETES_API_PROTOCOL", "http");
        String hostName = getEnvOrDefault("KUBERNETES_RO_SERVICE_HOST", "localhost
        String hostPort = getEnvOrDefault("KUBERNETES_RO_SERVICE_PORT", "8080");

        String host = protocol + "://" + hostName + ":" + hostPort;
        String serviceName = getEnvOrDefault("CASSANDRA_SERVICE", "cassandra");
        String path = "/api/v1beta1/endpoints/";
        try {
        URL url = new URL(host + path + serviceName);
        ObjectMapper mapper = new ObjectMapper();
        Endpoints endpoints = mapper.readValue(url, Endpoints.class);
        if (endpoints != null) {
        for (String endpoint : endpoints.endpoints) {
            String[] parts = endpoint.split(":");
            list.add(InetAddress.getByName(parts[0]));
        }
        }
        } catch (IOException ex) {
        logger.warn("Request to kubernetes apiserver failed");
        }
        if (list.size() == 0) {
        // If we got nothing, we might be the first instance, in that case
        // fall back on the seeds that were passed in cassandra.yaml.
        return defaultSeeds;
        }
        return list;
    }

    // Simple main to test the implementation
    public static void main(String[] args) {
        SeedProvider provider = new KubernetesSeedProvider(new HashMap<String, Str
        System.out.println(provider.getSeeds());
    }
}
```

GuestBook example

This example shows how to build a simple multi-tier web application using Kubernetes and Docker.

The example combines a web frontend, a redis master for storage and a replicated set of redis slaves.

Step Zero: Prerequisites

This example assumes that you have a basic understanding of kubernetes services and that you have forked the repository and [turned up a Kubernetes cluster]():

```
$ cd kubernetes
$ hack/dev-build-and-up.sh
```

Step One: Turn up the redis master

Use the file `examples/guestbook/redis-master.json` which describes a single pod running a redis key-value server in a container:

```
{
  "id": "redis-master",
  "kind": "Pod",
  "apiVersion": "v1beta1",
  "desiredState": {
    "manifest": {
      "version": "v1beta1",
      "id": "redis-master",
      "containers": [{
        "name": "master",
        "image": "dockerfile/redis",
        "cpu": 100,
        "ports": [{
          "containerPort": 6379,
          "hostPort": 6379
        }]
      }]
    }
  },
  "labels": {
    "name": "redis-master"
  }
}
```

Create the redis pod in your Kubernetes cluster by running:

```
$ cluster/kubectl.sh create -f examples/guestbook/redis-master.json
```

Once that's up you can list the pods in the cluster, to verify that the master is running:

```
cluster/kubectl.sh get pods
```

You'll see a single redis master pod. It will also display the machine that the pod is running on once it gets placed (may take up to thirty seconds):

```
NAME                IMAGE(S)            HOST
redis-master        dockerfile/redis    kubernetes-minion-2.c.myproject.internal/1
```

If you ssh to that machine, you can run `docker ps` to see the actual pod:

```
me@workstation$ gcloud compute ssh --zone us-central1-b kubernetes-minion-2
```

```
me@kubernetes-minion-2:~$ sudo docker ps
CONTAINER ID  IMAGE                     COMMAND                CREATED         STA
e3eed3e5e6d1  dockerfile/redis:latest   "redis-server /etc/re  2 minutes ago   Up
```

(Note that initial `docker pull` may take a few minutes, depending on network conditions. During this time, the `get pods` command will return `Pending` because the container has not yet started)

Step Two: Turn up the master service

A Kubernetes 'service' is a named load balancer that proxies traffic to one or more containers. The services in a Kubernetes cluster are discoverable inside other containers via *environment variables*. Services find the containers to load balance based on pod labels. These environment variables are typically referenced in application code, shell scripts, or other places where one node needs to talk to another in a distributed system. You should catch up on kubernetes services before proceeding.

The pod that you created in Step One has the label `name=redis-master`. The selector field of the service determines which pods will receive the traffic sent to the service. Use the file `examples/guestbook/redis-master-service.json`:

```
{
  "id": "redis-master",
  "kind": "Service",
  "apiVersion": "v1beta1",
  "port": 6379,
  "containerPort": 6379,
  "selector": {
    "name": "redis-master"
  },
  "labels": {
    "name": "redis-master"
  }
}
```

to create the service by running:

```
$ cluster/kubectl.sh create -f examples/guestbook/redis-master-service.json
redis-master

$ cluster/kubectl.sh get services
NAME                LABELS              SELECTOR
kubernetes          <none>              component=apiserver,provider=kubernetes
kubernetes-ro       <none>              component=apiserver,provider=kubernetes
redis-master        name=redis-master   name=redis-master
```

This will cause all pods to see the redis master apparently running on :6379.

Once created, the service proxy on each minion is configured to set up a proxy on the specified port (in this case port 6379).

Step Three: Turn up the replicated slave pods

Although the redis master is a single pod, the redis read slaves are a 'replicated' pod. In Kubernetes, a replication controller is responsible for managing multiple instances of a replicated pod.

Use the file `examples/guestbook/redis-slave-controller.json`:

```
{
  "id": "redis-slave-controller",
  "kind": "ReplicationController",
```

```
  "apiVersion": "v1beta1",
  "desiredState": {
    "replicas": 2,
    "replicaSelector": {"name": "redisslave"},
    "podTemplate": {
      "desiredState": {
         "manifest": {
           "version": "v1beta1",
           "id": "redis-slave-controller",
           "containers": [{
             "name": "slave",
             "image": "brendanburns/redis-slave",
             "cpu": 200,
             "ports": [{"containerPort": 6379, "hostPort": 6380}]
           }]
         }
       },
       "labels": {
         "name": "redisslave",
         "uses": "redis-master",
       }
      }},
  "labels": {"name": "redisslave"}
}
```

to create the replication controller by running:

```
$ cluster/kubectl.sh create -f examples/guestbook/redis-slave-controller.json
redis-slave-controller

# cluster/kubectl.sh get replicationcontrollers
NAME                    IMAGE(S)                   SELECTOR            REPLICAS
redis-slave-controller  brendanburns/redis-slave   name=redisslave     2
```

The redis slave configures itself by looking for the Kubernetes service environment variables in the container environment. In particular, the redis slave is started with the following command:

```
redis-server --slaveof ${REDIS_MASTER_SERVICE_HOST:-$SERVICE_HOST} $REDIS_MASTER_S
```

You might be curious about where the *REDIS_MASTER_SERVICE_HOST* is coming from. It is provided to this container when it is launched via the kubernetes services, which create environment variables (there is a well defined syntax for how service names get transformed to environment variable names in the documentation linked above).

Once that's up you can list the pods in the cluster, to verify that the master and slaves are running:

```
$ cluster/kubectl.sh get pods
NAME                                    IMAGE(S)                   HOST
redis-master                            dockerfile/redis           kubernetes-minio
ee68394b-7fca-11e4-a220-42010af0a5f1    brendanburns/redis-slave   kubernetes-minio
ee694768-7fca-11e4-a220-42010af0a5f1    brendanburns/redis-slave   kubernetes-minio
```

You will see a single redis master pod and two redis slave pods.

Step Four: Create the redis slave service

Just like the master, we want to have a service to proxy connections to the read slaves. In this case, in addition to discovery, the slave service provides transparent load balancing to clients. The service specification for the slaves is in `examples/guestbook/redis-slave-service.json`:

```
{
  "id": "redisslave",
  "kind": "Service",
  "apiVersion": "v1beta1",
  "port": 6379,
  "containerPort": 6379,
  "labels": {
    "name": "redisslave"
  },
  "selector": {
    "name": "redisslave"
  }
}
```

This time the selector for the service is `name=redisslave`, because that identifies the pods running redis slaves. It may also be helpful to set labels on your service itself as we've done here to make it easy to locate them with the `cluster/kubectl.sh get services -l "label=value"` command.

Now that you have created the service specification, create it in your cluster by running:

```
$ cluster/kubectl.sh create -f examples/guestbook/redis-slave-service.json
redisslave

$ cluster/kubectl.sh get services

NAME                LABELS              SELECTOR
kubernetes          <none>              component=apiserver,provider=kubernetes
kubernetes-ro       <none>              component=apiserver,provider=kubernetes
redis-master        name=redis-master   name=redis-master
redisslave          name=redisslave     name=redisslave
```

Step Five: Create the frontend pod

This is a simple PHP server that is configured to talk to either the slave or master services depending on whether the request is a read or a write. It exposes a simple AJAX interface, and serves an angular-based UX. Like the redis read slaves it is a replicated service instantiated by a replication controller.

The pod is described in the file `examples/guestbook/frontend-controller.json`:

```
{
  "id": "frontend-controller",
  "kind": "ReplicationController",
  "apiVersion": "v1beta1",
  "desiredState": {
    "replicas": 3,
    "replicaSelector": {"name": "frontend"},
    "podTemplate": {
      "desiredState": {
         "manifest": {
           "version": "v1beta1",
           "id": "frontend-controller",
           "containers": [{
             "name": "php-redis",
             "image": "kubernetes/example-guestbook-php-redis",
             "cpu": 100,
             "memory": 50000000,
             "ports": [{"containerPort": 80, "hostPort": 8000}]
           }]
         }
       },
```

```
       "labels": {
         "name": "frontend",
         "uses": "redisslave,redis-master"
       }
      }},
  "labels": {"name": "frontend"}
}
```

Using this file, you can turn up your frontend with:

```
$ cluster/kubectl.sh create -f examples/guestbook/frontend-controller.json
frontend-controller

$ cluster/kubectl.sh get replicationcontrollers
NAME                    IMAGE(S)                                SELECTOR
redis-slave-controller  brendanburns/redis-slave                name=redisslave
frontend-controller     kubernetes/example-guestbook-php-redis  name=frontend
```

Once that's up (it may take ten to thirty seconds to create the pods) you can list the pods in the cluster, to verify that the master, slaves and frontends are running:

```
$ cluster/kubectl.sh get pods
NAME                                    IMAGE(S)
redis-master                            dockerfile/redis
ee68394b-7fca-11e4-a220-42010af0a5f1    brendanburns/redis-slave
ee694768-7fca-11e4-a220-42010af0a5f1    brendanburns/redis-slave
9fbad0d6-7fcb-11e4-a220-42010af0a5f1    kubernetes/example-guestbook-php-redis
9fbbf70e-7fcb-11e4-a220-42010af0a5f1    kubernetes/example-guestbook-php-redis
9fbdbeca-7fcb-11e4-a220-42010af0a5f1    kubernetes/example-guestbook-php-redis
```

You will see a single redis master pod, two redis slaves, and three frontend pods.

The code for the PHP service looks like this:

```
<?

set_include_path('.:/usr/share/php:/usr/share/pear:/vendor/predis');

error_reporting(E_ALL);
ini_set('display_errors', 1);

require 'predis/autoload.php';

if (isset($_GET['cmd']) === true) {
  header('Content-Type: application/json');
  if ($_GET['cmd'] == 'set') {
    $client = new Predis\Client([
      'scheme' => 'tcp',
      'host'   => getenv('REDIS_MASTER_SERVICE_HOST') ?: getenv('SERVICE_HOST'),
      'port'   => getenv('REDIS_MASTER_SERVICE_PORT'),
    ]);
    $client->set($_GET['key'], $_GET['value']);
    print('{"message": "Updated"}');
  } else {
    $read_port = getenv('REDIS_MASTER_SERVICE_PORT');

    if (isset($_ENV['REDISSLAVE_SERVICE_PORT'])) {
      $read_port = getenv('REDISSLAVE_SERVICE_PORT');
    }
    $client = new Predis\Client([
```

```
      'scheme' => 'tcp',
      'host'   => getenv('REDIS_MASTER_SERVICE_HOST') ?: getenv('SERVICE_HOST'),
      'port'   => $read_port,
    ]);

    $value = $client->get($_GET['key']);
    print('{"data": "' . $value . '"}');
  }
} else {
  phpinfo();
} ?>
```

To play with the service itself, find the name of a frontend, grab the external IP of that host from the Google Cloud Console or the `gcloud` tool, and visit `http://<host-ip>:8000`.

```
$ gcloud compute instances list
```

You may need to open the firewall for port 8000 using the console or the `gcloud` tool. The following command will allow traffic from any source to instances tagged `kubernetes-minion`:

```
$ gcloud compute firewall-rules create --allow=tcp:8000 --target-tags=kubernetes-r
```

If you are running Kubernetes locally, you can just visit http://localhost:8000. For details about limiting traffic to specific sources, see the GCE firewall documentation.

Step Six: Cleanup

To turn down a Kubernetes cluster:

```
$ cluster/kube-down.sh
```

Building and releasing Guestbook Image

This process employs building two docker images, one compiles the source and the other hosts the compiled binaries.

Releasing the image requires that you have access to the docker registry user account which will host the image.

To build and release the guestbook image:

```
cd examples/guestbook-go/_src
./script/release.sh
```

Step by step

If you may want to, you can build and push the image step by step.

Start fresh before building

```
./script/clean.sh 2> /dev/null
```

Build

Builds a docker image that builds the app and packages it into a minimal docker image

```
./script/build.sh
```

Push

Accepts an optional tag (defaults to "latest")

```
./script/push.sh [TAG]
```

Clean up

```
./script/clean.sh
```

GuestBook example

This example shows how to build a simple multi-tier web application using Kubernetes and Docker.

The example combines a web frontend, a redis master for storage and a replicated set of redis slaves.

Step Zero: Prerequisites

This example assumes that you have forked the repository and turned up a Kubernetes cluster:

```
$ cd kubernetes
$ hack/dev-build-and-up.sh
```

Step One: Turn up the redis master.

Use the file `examples/guestbook-go/redis-master-controller.json` to create a replication controller which manages a single pod. The pod runs a redis key-value server in a container. Using a replication controller is the preferred way to launch long-running pods, even for 1 replica, so the pod will benefit from self-healing mechanism in kubernetes.

Create the redis master replication controller in your Kubernetes cluster using the `kubectl` CLI:

```
$ cluster/kubectl.sh create -f examples/guestbook-go/redis-master-controller.json
```

Once that's up you can list the replication controllers in the cluster:

```
$ cluster/kubectl.sh get rc
CONTROLLER                     CONTAINER(S)            IMAGE(S)
redis-master-controller        redis-master            gurpartap/redis
```

List pods in cluster to verify the master is running. You'll see a single redis master pod. It will also display the machine that the pod is running on once it gets placed (may take up to thirty seconds).

```
$ cluster/kubectl.sh get pods
POD                       IP                  CONTAINER(S)        IMAGE(S)
redis-master-pod-hh2gd    10.244.3.7          redis-master        gurpartap/redis
```

If you ssh to that machine, you can run `docker ps` to see the actual pod:

```
me@workstation$ gcloud compute ssh --zone us-central1-b kubernetes-minion-4

me@kubernetes-minion-3:~$ sudo docker ps
CONTAINER ID        IMAGE                                COMMAND
d5c458dabe50        gurpartap/redis:latest               "/usr/local/bin/redi
```

(Note that initial `docker pull` may take a few minutes, depending on network conditions.)

Step Two: Turn up the master service.

A Kubernetes 'service' is a named load balancer that proxies traffic to one or more containers. The services in a Kubernetes cluster are discoverable inside other containers via environment variables. Services find the containers to load balance based on pod labels.

The pod that you created in Step One has the label `name=redis` and `role=master`. The selector field of the service determines which pods will receive the traffic sent to the service. Use the file `examples/guestbook-go/redis-master-service.json` to create the service in the `kubectl` cli:

```
$ cluster/kubectl.sh create -f examples/guestbook-go/redis-master-service.json

$ cluster/kubectl.sh get services
```

```
NAME                    LABELS                                  SELECTOR
redis-master            <none>                                  name=redis,role=
```

This will cause all new pods to see the redis master apparently running on $REDIS_MASTER_SERVICE_HOST at port 6379. Once created, the service proxy on each node is configured to set up a proxy on the specified port (in this case port 6379).

Step Three: Turn up the replicated slave pods.

Although the redis master is a single pod, the redis read slaves are a 'replicated' pod. In Kubernetes, a replication controller is responsible for managing multiple instances of a replicated pod.

Use the file `examples/guestbook-go/redis-slave-controller.json` to create the replication controller:

```
$ cluster/kubectl.sh create -f examples/guestbook-go/redis-slave-controller.json

$ cluster/kubectl.sh get rc
CONTROLLER                      CONTAINER(S)            IMAGE(S)
redis-master-controller         redis-master            gurpartap/redis
redis-slave-controller          redis-slave             gurpartap/redis
```

The redis slave configures itself by looking for the Kubernetes service environment variables in the container environment. In particular, the redis slave is started with the following command:

```
redis-server --slaveof $REDIS_MASTER_SERVICE_HOST $REDIS_MASTER_SERVICE_PORT
```

Once that's up you can list the pods in the cluster, to verify that the master and slaves are running:

```
$ cluster/kubectl.sh get pods
POD                             IP                  CONTAINER(S)
redis-master-pod-hh2gd          10.244.3.7          redis-master
redis-slave-controller-i7hvs    10.244.2.7          redis-slave
redis-slave-controller-nyxxv    10.244.1.6          redis-slave
```

You will see a single redis master pod and two redis slave pods.

Step Four: Create the redis slave service.

Just like the master, we want to have a service to proxy connections to the read slaves. In this case, in addition to discovery, the slave service provides transparent load balancing to clients. The service specification for the slaves is in `examples/guestbook-go/redis-slave-service.json`

This time the selector for the service is `name=redis,role=slave`, because that identifies the pods running redis slaves. It may also be helpful to set labels on your service itself--as we've done here--to make it easy to locate them later.

Now that you have created the service specification, create it in your cluster with the `kubectl` CLI:

```
$ cluster/kubectl.sh create -f examples/guestbook-go/redis-slave-service.json

$ cluster/kubectl.sh get services
NAME                    LABELS                                  SELECTOR
redis-master            <none>                                  name=redis,role=
redis-slave             name=redis,role=slave                   name=redis,role=
```

Step Five: Create the guestbook pod.

This is a simple Go net/http (negroni based) server that is configured to talk to either the slave or master services depending on whether the request is a read or a write. It exposes a simple JSON interface, and

serves a jQuery-Ajax based UX. Like the redis read slaves it is a replicated service instantiated by a replication controller.

The pod is described in the file `examples/guestbook-go/guestbook-controller.json`. Using this file, you can turn up your guestbook with:

```
$ cluster/kubectl.sh create -f examples/guestbook-go/guestbook-controller.json

$ cluster/kubectl.sh get replicationControllers
CONTROLLER                     CONTAINER(S)            IMAGE(S)
guestbook-controller           guestbook               kubernetes/guestboo
redis-master-controller        redis-master            gurpartap/redis
redis-slave-controller         redis-slave             gurpartap/redis
```

Once that's up (it may take ten to thirty seconds to create the pods) you can list the pods in the cluster, to verify that the master, slaves and guestbook frontends are running:

```
$ cluster/kubectl.sh get pods
POD                                  IP                  CONTAINER(S)
guestbook-controller-182tv           10.244.2.8          guestbook
guestbook-controller-jzjpe           10.244.0.7          guestbook
guestbook-controller-zwk1b           10.244.3.8          guestbook
redis-master-pod-hh2gd               10.244.3.7          redis-master
redis-slave-controller-i7hvs         10.244.2.7          redis-slave
redis-slave-controller-nyxxv         10.244.1.6          redis-slave
```

You will see a single redis master pod, two redis slaves, and three guestbook pods.

Step Six: Create the guestbook service.

Just like the others, you want a service to group your guestbook pods. The service specification for the guestbook is in `examples/guestbook-go/guestbook-service.json`. There's a twist this time - because we want it to be externally visible, we set the `createExternalLoadBalancer` flag on the service.

```
$ cluster/kubectl.sh create -f examples/guestbook-go/guestbook-service.json

$ cluster/kubectl.sh get services
NAME                    LABELS                                    SELECTOR
guestbook               <none>                                    name=guestbook
redis-master            <none>                                    name=redis,role=
redis-slave             name=redis,role=slave                     name=redis,role=
```

To play with the service itself, find the external IP of the load balancer from the Google Cloud Console or the `gcloud` tool, and visit `http://<ip>:3000`.

```
$ gcloud compute forwarding-rules describe --region=us-central1 guestbook
IPAddress: 11.22.33.44
IPProtocol: TCP
creationTimestamp: '2014-11-24T16:08:15.327-08:00'
id: '17594840560849468061'
kind: compute#forwardingRule
name: guestbook
portRange: 1-65535
region: https://www.googleapis.com/compute/v1/projects/jbeda-prod/regions/us-centr
selfLink: https://www.googleapis.com/compute/v1/projects/jbeda-prod/regions/us-cen
target: https://www.googleapis.com/compute/v1/projects/jbeda-prod/regions/us-centr
```

You may need to open the firewall for port 3000 using the console or the `gcloud` tool. The following command will allow traffic from any source to instances tagged `kubernetes-minion`:

```
$ gcloud compute firewall-rules create --allow=tcp:3000 --target-tags=kubernetes-r
```

If you are running Kubernetes locally, you can just visit http://localhost:3000 For details about limiting traffic to specific sources, see the GCE firewall documentation.

Step Seven: Cleanup

To turn down a Kubernetes cluster:

```
$ cluster/kube-down.sh
```

Kubernetes Namespaces

Kubernetes Namespaces help different projects, teams, or customers to share a Kubernetes cluster.

It does this by providing the following:

1. A scope for Names.
2. A mechanism to attach authorization and policy to a subsection of the cluster.

Use of multiple namespaces is optional.

This example demonstrates how to use Kubernetes namespaces to subdivide your cluster.

Step Zero: Prerequisites

This example assumes the following:

1. You have an existing Kubernetes cluster.
2. You have a basic understanding of Kubernetes pods, services, and replication controllers.

Step One: Understand the default namespace

By default, a Kubernetes cluster will instantiate a default namespace when provisioning the cluster to hold the default set of pods, services, and replication controllers used by the cluster.

Assuming you have a fresh cluster, you can introspect the available namespace's by doing the following:

```
$ cluster/kubectl.sh get namespaces
NAME                LABELS
default             <none>
```

Step Two: Create new namespaces

For this exercise, we will create two additional Kubernetes namespaces to hold our content.

Let's imagine a scenario where an organization is using a shared Kubernetes cluster for development and production use cases.

The development team would like to maintain a space in the cluster where they can get a view on the list of pods, services, and replication-controllers they use to build and run their application. In this space, Kubernetes resources come and go, and the restrictions on who can or cannot modify resources are relaxed to enable agile development.

The operations team would like to maintain a space in the cluster where they can enforce strict procedures on who can or cannot manipulate the set of pods, services, and replication controllers that run the production site.

One pattern this organization could follow is to partition the Kubernetes cluster into two namespaces: development and production.

Let's create two new namespaces to hold our work.

Use the file `examples/kubernetes-namespaces/namespace-dev.json` which describes a development namespace:

```
{
  "kind": "Namespace",
  "apiVersion":"v1beta1",
  "id": "development",
  "spec": {},
```

```
  "status": {},
  "labels": {
    "name": "development"
  },
}
```

Create the development namespace using kubectl.

```
$ cluster/kubectl.sh create -f examples/kubernetes-namespaces/namespace-dev.json
```

And then lets create the production namespace using kubectl.

```
$ cluster/kubectl.sh create -f examples/kubernetes-namespaces/namespace-prod.json
```

To be sure things are right, let's list all of the namespaces in our cluster.

```
$ cluster/kubectl.sh get namespaces
NAME                LABELS
default             <none>
development         name=development
production          name=production
```

Step Three: Create pods in each namespace

A Kubernetes namespace provides the scope for pods, services, and replication controllers in the cluster.

Users interacting with one namespace do not see the content in another namespace.

To demonstrate this, let's spin up a simple replication controller and pod in the development namespace.

The first step is to define a context for the kubectl client to work in each namespace.

```
$ cluster/kubectl.sh config set-context dev --namespace=development
$ cluster/kubectl.sh config set-context prod --namespace=production
```

The above commands provided two request contexts you can alternate against depending on what namespace you wish to work against.

Let's switch to operate in the development namespace.

```
$ cluster/kubectl.sh config use-context dev
```

You can verify your current context by doing the following:

```
$ cluster/kubectl.sh config view
clusters: {}
contexts:
  dev:
    cluster: ""
    namespace: development
    user: ""
  prod:
    cluster: ""
    namespace: production
    user: ""
current-context: dev
preferences: {}
users: {}
```

At this point, all requests we make to the Kubernetes cluster from the command line are scoped to the development namespace.

Let's create some content.

```
$ cluster/kubectl.sh run-container snowflake --image=kubernetes/serve_hostname --r
```

We have just created a replication controller whose replica size is 2 that is running the pod called snowflake with a basic container that just serves the hostname.

```
cluster/kubectl.sh get rc
CONTROLLER          CONTAINER(S)        IMAGE(S)                    SELECTOR
snowflake           snowflake           kubernetes/serve_hostname   run-container=

$ cluster/kubectl.sh get pods
POD                 IP                  CONTAINER(S)        IMAGE(S)
snowflake-fplln     10.246.0.5          snowflake           kubernetes/serve_hostn
snowflake-gziey     10.246.0.4          snowflake           kubernetes/serve_hostn
```

And this is great, developers are able to do what they want, and they do not have to worry about affecting content in the production namespace.

Let's switch to the production namespace and show how resources in one namespace are hidden from the other.

```
$ cluster/kubectl.sh config use-context prod
```

The production namespace should be empty.

```
$ cluster/kubectl.sh get rc
CONTROLLER          CONTAINER(S)        IMAGE(S)            SELECTOR            RE

$ cluster/kubectl.sh get pods
POD                 IP                  CONTAINER(S)        IMAGE(S)            HO
```

Production likes to run cattle, so let's create some cattle pods.

```
$ cluster/kubectl.sh run-container cattle --image=kubernetes/serve_hostname --repl

$ cluster/kubectl.sh get rc
CONTROLLER          CONTAINER(S)        IMAGE(S)                    SELECTOR
cattle              cattle              kubernetes/serve_hostname   run-container=

$ cluster/kubectl.sh get pods
POD                 IP                  CONTAINER(S)        IMAGE(S)
cattle-0133o        10.246.0.7          cattle              kubernetes/serve_hostn
cattle-hh2gd        10.246.0.10         cattle              kubernetes/serve_hostn
cattle-ls6k1        10.246.0.9          cattle              kubernetes/serve_hostn
cattle-nyxxv        10.246.0.8          cattle              kubernetes/serve_hostn
cattle-oh43e        10.246.0.6          cattle              kubernetes/serve_hostn
```

At this point, it should be clear that the resources users create in one namespace are hidden from the other namespace.

As the policy support in Kubernetes evolves, we will extend this scenario to show how you can provide different authorization rules for each namespace.

Node selection example

This example shows how to assign a pod to a specific node or to one of a set of nodes using node labels and the nodeSelector field in a pod specification. Generally this is unnecessary, as the scheduler will take care of things for you, but you may want to do so in certain circumstances like to ensure that your pod ends up on a machine with an SSD attached to it.

Step Zero: Prerequisites

This example assumes that you have a basic understanding of kubernetes pods and that you have turned up a Kubernetes cluster.

Step One: Attach label to the node

Run `kubectl get nodes` to get the names of the nodes. Pick out the one that you want to add a label to. Note that label keys must be in the form of DNS labels (as described in the identifiers doc), meaning that they are not allowed to contain any upper-case letters. Then run `kubectl get node <node-name> -o yaml > node.yaml`. The contents of the file should look something like this:

```
apiVersion: v1beta1
creationTimestamp: 2015-02-03T01:16:46Z
hostIP: 104.154.60.112
id:
kind: Node
resourceVersion: 12
resources:
  capacity:
    cpu: "1"
    memory: 4.0265318e+09
selfLink: /api/v1beta1/minions/
status:
  conditions:
  - kind: Ready
    lastTransitionTime: null
    status: Full
uid: 526a4156-ab42-11e4-9817-42010af0258d
```

Add the labels that you want to the file like this:

```
apiVersion: v1beta1
creationTimestamp: 2015-02-03T01:16:46Z
hostIP: 104.154.60.112
id:
kind: Node
labels:
  disktype: ssd
resourceVersion: 12
resources:
  capacity:
    cpu: "1"
    memory: 4.0265318e+09
selfLink: /api/v1beta1/minions/
status:
  conditions:
  - kind: Ready
    lastTransitionTime: null
    status: Full
uid: 526a4156-ab42-11e4-9817-42010af0258d
```

Then update the node by running `kubectl update -f node.yaml`. Make sure that the resourceVersion you use in your update call is the same as the resourceVersion returned by the get call. If something about the node changes between your get and your update, the update will fail because the resourceVersion will have changed.

Note that as of 2015-02-03 there are a couple open issues that prevent this from working without modification. Due to issue #3005, you have to remove all status-related fields from the file, which is both everything under the `status` field as well as the `hostIP` field (removing hostIP isn't required in v1beta3). Due to issue 4041, you may have to modify the representation of the resource capacity numbers to make them integers. These are both temporary, and fixes are being worked on. In the meantime, you would actually call `kubectl update -f node.yaml` with a file that looks like this:

```
apiVersion: v1beta1
creationTimestamp: 2015-02-03T01:16:46Z
id:
kind: Node
labels:
  disktype: ssd
resourceVersion: 12
resources:
  capacity:
    cpu: "1"
    memory: 4026531800
selfLink: /api/v1beta1/minions/
uid: 526a4156-ab42-11e4-9817-42010af0258d
```

Step Two: Add a nodeSelector field to your pod configuration

Take whatever pod config file you want to run, and add a nodeSelector section to it, like this. For example, if this is my pod config:

```
apiVersion: v1beta1
desiredState:
  manifest:
    containers:
      - image: nginx
        name: nginx
    id: nginx
    version: v1beta1
id: nginx
kind: Pod
labels:
  env: test
```

Then add a nodeSelector like so:

```
apiVersion: v1beta1
desiredState:
  manifest:
    containers:
      - image: nginx
        name: nginx
    id: nginx
    version: v1beta1
id: nginx
kind: Pod
labels:
  env: test
nodeSelector:
  disktype: ssd
```

When you then run `kubectl create -f pod.yaml`, the pod will get scheduled on the node that you attached the label to! You can verify that it worked by running `kubectl get pods` and looking at the "host" that the pod was assigned to.

Conclusion

While this example only covered one node, you can attach labels to as many nodes as you want. Then when you schedule a pod with a nodeSelector, it can be scheduled on any of the nodes that satisfy that nodeSelector. Be careful that it will match at least one node, however, because if it doesn't the pod won't be scheduled at all.

RethinkDB Cluster on Kubernetes

Setting up a rethinkdb cluster on kubernetes

Features

- Auto configuration cluster by querying info from k8s
- Simple

Quick start

Step 1

antmanler/rethinkdb will discover peer using endpoints provided by kubernetes_ro service, so first create a service so the following pod can query its endpoint

```
kubectl create -f driver-service.yaml
```

check out:

```
$kubectl get se
NAME                LABELS              SELECTOR                  IP
rethinkdb-driver    db=influxdb         db=rethinkdb              10.241.105.47
```

Step 2

start fist server in cluster

```
kubectl create -f rc.yaml
```

Actually, you can start servers as many as you want at one time, just modify the `replicas` in `rc.ymal`

check out again:

```
$kubectl get po
POD                                   IP                  CONTAINER(S)        IMA
99f6d361-abd6-11e4-a1ea-001c426dbc28  10.240.2.68         rethinkdb           ret
```

Done!

Scale

You can scale up you cluster using `kubectl resize`, and new pod will join to exsits cluster automatically, for example

```
$kubectl resize rc rethinkdb-rc-1.16.0 --replicas=3
resized
$kubectl get po
POD                                    IP                  CONTAINER(S)        IMA
99f6d361-abd6-11e4-a1ea-001c426dbc28   10.240.2.68         rethinkdb           ret
d10182b5-abd6-11e4-a1ea-001c426dbc28   10.240.26.14        rethinkdb           ret
d101c1a4-abd6-11e4-a1ea-001c426dbc28   10.240.11.14        rethinkdb           ret
```

Admin

You need a separate pod (which labled as role:admin) to access Web Admin UI

```
kubectl create -f admin-pod.yaml
kubectl create -f admin-service.yaml
```

find the service

```
$kubectl get se
NAME                LABELS              SELECTOR                    IP
rethinkdb-admin     db=influxdb         db=rethinkdb,role=admin     10.241.220.209
rethinkdb-driver    db=influxdb         db=rethinkdb                10.241.105.47
```

open a web browser and access to *http:///10.241.220.209:8080* to manage you cluster

Why not just using pods in replicas?

This is because kube-proxy will act as a load balancer and send your traffic to different server, since the ui is not stateless when playing with Web Admin UI will cause `Connection not open on server` error.

BTW

- All services and pods are placed under namespace `rethinkdb`.
- `gen_pod.sh` is using to generate pod templates for my local cluster, the generated pods which is using `nodeSelector` to force k8s to schedule containers to my designate nodes, for I need to access persistent data on my host dirs.
- see [antmanler/rethinkdb-k8s](#) for detail

Live update example

This example demonstrates the usage of Kubernetes to perform a live update on a running group of pods.

Step Zero: Prerequisites

This example assumes that you have forked the repository and turned up a Kubernetes cluster:

```
$ cd kubernetes
$ hack/dev-build-and-up.sh
```

This example also assumes that you have Docker installed on your local machine.

It also assumes that `$DOCKER_HUB_USER` is set to your Docker user id. We use this to upload the docker images that are used in the demo.

```
$ export DOCKER_HUB_USER=my-docker-id
```

You may need to open the firewall for port 8080 using the console or the `gcloud` tool. The following command will allow traffic from any source to instances tagged `kubernetes-minion`:

```
$ gcloud compute firewall-rules create \
  --allow tcp:8080 --target-tags=kubernetes-minion \
  --zone=us-central1-a kubernetes-minion-8080
```

Step Zero: Build the Docker images

This can take a few minutes to download/upload stuff.

```
$ cd examples/update-demo
$ ./0-build-images.sh
```

Step One: Turn up the UX for the demo

You can use bash job control to run this in the background. This can sometimes spew to the output so you could also run it in a different terminal.

```
$ ./1-run-web-proxy.sh &
Running local proxy to Kubernetes API Server.  Run this in a
separate terminal or run it in the background.

    http://localhost:8001/static/

+ ../../cluster/kubectl.sh proxy --www=local/
I0115 16:50:15.959551   19790 proxy.go:34] Starting to serve on localhost:8001
```

Now visit the the demo website. You won't see anything much quite yet.

Step Two: Run the controller

Now we will turn up two replicas of an image. They all serve on port 8080, mapped to internal port 80

```
$ ./2-create-replication-controller.sh
```

After pulling the image from the Docker Hub to your worker nodes (which may take a minute or so) you'll see a couple of squares in the UI detailing the pods that are running along with the image that they are serving up. A cute little nautilus.

Step Three: Try resizing the controller

Now we will increase the number of replicas from two to four:

```
$ ./3-scale.sh
```

If you go back to the demo website you should eventually see four boxes, one for each pod.

Step Four: Update the docker image

We will now update the docker image to serve a different image by doing a rolling update to a new Docker image.

```
$ ./4-rolling-update.sh
```

The rollingUpdate command in kubectl will do 2 things:

1. Create a new replication controller with a pod template that uses the new image (`$DOCKER_HUB_USER/update-demo:kitten`)
2. Resize the old and new replication controllers until the new controller replaces the old. This will kill the current pods one at a time, spinnning up new ones to replace them.

Watch the demo website, it will update one pod every 10 seconds until all of the pods have the new image.

Step Five: Bring down the pods

```
$ ./5-down.sh
```

This will first 'stop' the replication controller by turning the target number of replicas to 0. It'll then delete that controller.

Step Six: Cleanup

To turn down a Kubernetes cluster:

```
$ cd ../..  # Up to kubernetes.
$ cluster/kube-down.sh
```

Kill the proxy running in the background: After you are done running this demo make sure to kill it:

```
$ jobs
[1]+  Running                 ./1-run-web-proxy.sh &
$ kill %1
[1]+  Terminated: 15          ./1-run-web-proxy.sh
```

Image Copyright

Note that he images included here are public domain.

- kitten
- nautilus

Kubernetes 201 - Labels, Replication Controllers, Services and Health Checking

Overview

When we had just left off in the previous episode we had learned about pods, multiple containers and volumes. We'll now cover some slightly more advanced topics in Kubernetes, related to application productionization, deployment and scaling.

Labels

Having already learned about Pods and how to create them, you may be struck by an urge to create many, many pods. Please do! But eventually you will need a system to organize these pods into groups. The system for achieving this in Kubernetes is Labels. Labels are key-value pairs that are attached to each API object in Kubernetes. Label selectors can be passed along with a RESTful `list` request to the apiserver to retrieve a list of objects which match that label selector. For example:

```
cluster/kubectl.sh get pods -l name=nginx
```

Lists all pods who name label matches 'nginx'. Labels are discussed in detail elsewhere, but they are a core concept for two additional building blocks for Kubernetes, Replication Controllers and Services

Replication Controllers

OK, now you have an awesome, multi-container, labelled pod and you want to use it to build an application, you might be tempted to just start building a whole bunch of individual pods, but if you do that, a whole host of operational concerns pop up. For example: how will you scale the number of pods up or down and how will you ensure that all pods are homogenous?

Replication controllers are the objects to answer these questions. A replication controller combines a template for pod creation (a "cookie-cutter" if you will) and a number of desired replicas, into a single API object. The replica controller also contains a label selector that identifies the set of objects managed by the replica controller. The replica controller constantly measures the size of this set relative to the desired size, and takes action by creating or deleting pods. The design of replica controllers is discussed in detail elsewhere.

An example replica controller that instantiates two pods running nginx looks like:

```
id: nginx-controller
apiVersion: v1beta1
kind: ReplicationController
desiredState:
  replicas: 2
  # replicaSelector identifies the set of Pods that this
  # replicaController is responsible for managing
  replicaSelector:
    name: nginx
  # podTemplate defines the 'cookie cutter' used for creating
  # new pods when necessary
  podTemplate:
    desiredState:
      manifest:
        version: v1beta1
        id: nginx
        containers:
          - name: nginx
            image: dockerfile/nginx
            ports:
```

```
          - containerPort: 80
    # Important: these labels need to match the selector above
    # The api server enforces this constraint.
    labels:
      name: nginx
```

Services

Once you have a replicated set of pods, you need an abstraction that enables connectivity between the layers of your application. For example, if you have a replication controller managing your backend jobs, you don't want to have to reconfigure your front-ends whenever you re-scale your backends. Likewise, if the pods in your backends are scheduled (or rescheduled) onto different machines, you can't be required to re-configure your front-ends. In Kubernetes the Service API object achieves these goals. A Service basically combines an IP address and a label selector together to form a simple, static rallying point for connecting to a micro-service in your application.

For example, here is a service that balances across the pods created in the previous nginx replication controller example:

```
kind: Service
apiVersion: v1beta1
# must be a DNS compatible name
id: nginx-example
# the port that this service should serve on
port: 8000
# just like the selector in the replication controller,
# but this time it identifies the set of pods to load balance
# traffic to.
selector:
  name: nginx
# the container on each pod to connect to, can be a name
# (e.g. 'www') or a number (e.g. 80)
containerPort: 80
```

When created, each service is assigned a unique IP address. This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the service, and know that communication to the service will be automatically load-balanced out to some pod that is a member of the set identified by the label selector in the Service. Services are described in detail elsewhere.

Health Checking

When I write code it never crashes, right? Sadly the kubernetes issues list indicates otherwise...

Rather than trying to write bug-free code, a better approach is to use a management system to perform periodic health checking and repair of your application. That way, a system, outside of your application itself, is responsible for monitoring the application and taking action to fix it. It's important that the system be outside of the application, since of course, if your application fails, and the health checking agent is part of your application, it may fail as well, and you'll never know. In Kubernetes, the health check monitor is the Kubelet agent.

Low level process health-checking

The simplest form of health-checking is just process level health checking. The Kubelet constantly asks the Docker daemon if the container process is still running, and if not, the container process is restarted. In all of the Kubernetes examples you have run so far, this health checking was actually already enabled. It's on for every single container that runs in Kubernetes.

Application health-checking

However, in many cases, this low-level health checking is insufficient. Consider for example, the following code:

```
lockOne := sync.Mutex{}
lockTwo := sync.Mutex{}

go func() {
  lockOne.Lock();
  lockTwo.Lock();
  ...
}()

lockTwo.Lock();
lockOne.Lock();
```

This is a classic example of a problem in computer science known as "Deadlock". From Docker's perspective your application is still operating, the process is still running, but from your application's perspective, your code is locked up, and will never respond correctly.

To address this problem, Kubernetes supports user implemented application health-checks. These checks are performed by the Kubelet to ensure that your application is operating correctly for a definition of "correctly" that *you* provide.

Currently, there are three types of application health checks that you can choose from:

- HTTP Health Checks - The Kubelet will call a web hook. If it returns between 200 and 399, it is considered success, failure otherwise.
- Container Exec - The Kubelet will execute a command inside your container. If it returns "ok" it will be considered a success.
- TCP Socket - The Kubelet will attempt to open a socket to your container. If it can establish a connection, the container is considered healthy, if it can't it is considered a failure.

In all cases, if the Kubelet discovers a failure, the container is restarted.

The container health checks are configured in the "LivenessProbe" section of your container config. There you can also specify an "initialDelaySeconds" that is a grace period from when the container is started to when health checks are performed, to enable your container to perform any necessary initialization.

Here is an example config for a pod with an HTTP health check:

```
kind: Pod
apiVersion: v1beta1
desiredState:
  manifest:
    version: v1beta1
    id: php
    containers:
      - name: nginx
        image: dockerfile/nginx
        ports:
          - containerPort: 80
        # defines the health checking
        livenessProbe:
          # turn on application health checking
          enabled: true
          type: http
          # length of time to wait for a pod to initialize
          # after pod startup, before applying health checking
          initialDelaySeconds: 30
          # an http probe
```

```
        httpGet:
          path: /_status/healthz
          port: 8080
```

What's next?

For a complete application see the guestbook example.

Kubernetes 101 - Walkthrough

Pods

The first atom of Kubernetes is a *pod*. A pod is a collection of containers that are symbiotically grouped.

See pods for more details.

Intro

Trivially, a single container might be a pod. For example, you can express a simple web server as a pod:

```
apiVersion: v1beta1
kind: Pod
id: www
desiredState:
  manifest:
    version: v1beta1
    id: www
    containers:
      - name: nginx
        image: dockerfile/nginx
```

A pod definition is a declaration of a *desired state*. Desired state is a very important concept in the Kubernetes model. Many things present a desired state to the system, and it is Kubernetes' responsibility to make sure that the current state matches the desired state. For example, when you create a Pod, you declare that you want the containers in it to be running. If the containers happen to not be running (e.g. program failure, ...), Kubernetes will continue to (re-)create them for you in order to drive them to the desired state. This process continues until you delete the Pod.

See the design document for more details.

Volumes

Now that's great for a static web server, but what about persistent storage? We know that the container file system only lives as long as the container does, so we need more persistent storage. To do this, you also declare a `volume` as part of your pod, and mount it into a container:

```
apiVersion: v1beta1
kind: Pod
id: storage
desiredState:
  manifest:
    version: v1beta1
    id: storage
    containers:
      - name: redis
        image: dockerfile/redis
        volumeMounts:
            # name must match the volume name below
          - name: redis-persistent-storage
            # mount path within the container
            mountPath: /data/redis
    volumes:
      - name: redis-persistent-storage
        source:
          emptyDir: {}
```

Ok, so what did we do? We added a volume to our pod:

```
...
    volumes:
      - name: redis-persistent-storage
        source:
          emptyDir: {}
...
```

And we added a reference to that volume to our container:

```
...
        volumeMounts:
            # name must match the volume name below
          - name: redis-persistent-storage
            # mount path within the container
            mountPath: /data/redis
...
```

In Kubernetes, `emptyDir` Volumes live for the lifespan of the Pod, which is longer than the lifespan of any one container, so if the container fails and is restarted, our persistent storage will live on.

If you want to mount a directory that already exists in the file system (e.g. `/var/logs`) you can use the `hostDir` directive.

See volumes for more details.

Multiple Containers

Note: The examples below are syntactically correct, but some of the images (e.g. kubernetes/git-monitor) don't exist yet. We're working on turning these into working examples.

However, often you want to have two different containers that work together. An example of this would be a web server, and a helper job that polls a git repository for new updates:

```
apiVersion: v1beta1
kind: Pod
id: www
desiredState:
  manifest:
    version: v1beta1
    id: www
    containers:
      - name: nginx
        image: dockerfile/nginx
        volumeMounts:
          - name: www-data
            mountPath: /srv/www
            readOnly: true
      - name: git-monitor
        image: kubernetes/git-monitor
        env:
          - name: GIT_REPO
            value: http://github.com/some/repo.git
        volumeMounts:
          - name: www-data
            mountPath: /data
    volumes:
      - name: www-data
        source:
          emptyDir
```

Note that we have also added a volume here. In this case, the volume is mounted into both containers. It is marked `readOnly` in the web server's case, since it doesn't need to write to the directory.

Finally, we have also introduced an environment variable to the `git-monitor` container, which allows us to parameterize that container with the particular git repository that we want to track.

What's next?

Continue on to Kubernetes 201 or for a complete application see the guestbook example