Arvados - Bug #13964

crunch-dispatch-cloud spike

08/03/2018 06:23 PM - Peter Amstutz

Status:	Resolved	**Start date:**	
Priority:	Normal	**Due date:**	
Assigned To:	Tom Clegg	**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:	2018-10-17 sprint		

Description

https://dev.arvados.org/projects/arvados/wiki/Dispatching_containers_to_cloud_VMs

Related issues:

Related to Arvados - Story #13908: [Epic] Replace SLURM for cloud job schedul...	**Resolved**	
Related to Arvados - Feature #14325: [crunch-dispatch-cloud] Dispatch contain...	**Resolved**	**01/28/2019**

History

#1 - 08/03/2018 06:23 PM - Peter Amstutz

- Status changed from New to In Progress

#2 - 08/03/2018 06:24 PM - Peter Amstutz

- Description updated

#3 - 08/04/2018 08:22 PM - Peter Amstutz

- Description updated

#4 - 08/15/2018 04:07 PM - Tom Morris

- Target version set to 2018-09-05 Sprint

#5 - 08/15/2018 04:08 PM - Peter Amstutz

pseudo code prototype 13964-cdc

#6 - 08/15/2018 04:24 PM - Tom Clegg

- Assigned To set to Tom Clegg

#7 - 08/17/2018 05:29 PM - Tom Clegg

```
package dispatchcloud

// A dispatcher comprises a container queue, a scheduler, a worker
// pool, a cloud provider, a stale-lock fixer, and a syncer.
// 1. Choose a provider.
// 2. Start a worker pool.
// 3. Start a container queue.
// 4. Run a stale-lock fixer.
// 5. Start a scheduler.
// 6. Start a syncer.
//
//
// A provider (cloud driver) creates new cloud VM instances and gets
// the latest list of instances. The returned instances implement
// proxies to the provider's metadata and control interfaces (get IP
// address, update tags, shutdown).
//
//
// A commander sets up an SSH connection to an instance, and
// retries/reconnects when needed, so it is ready to execute remote
// commands on behalf of callers.
//
//
// A workerPool tracks workers' instance types and readiness states
```

```
// (available to do work now, booting, suffering a temporary network
// outage, shutting down). It loads internal state from the cloud
// provider's list of instances at startup, and syncs periodically
// after that.
//
//
// A worker manages communication with a cloud instance, and notifies
// the worker pool when it becomes available to do work after booting,
// finishing a container, emerging from a network outage, etc.
//
//
// A container queue tracks the known state (according to
// arvados-controller) of each container of interest -- i.e., queued,
// or locked/running using our own dispatch token. It also proxies the
// dispatcher's lock/unlock/cancel requests to the controller. It
// handles concurrent refresh and update operations without exposing
// out-of-order updates to its callers. (It drops any new information
// that might have originated before its own most recent
// lock/unlock/cancel operation.)
//
//
// A stale-lock fixer waits for any already-locked containers (i.e.,
// locked by a prior server process) to appear on workers as the
// worker pool recovers its state. It unlocks/requeues any that still
// remain when all workers are recovered or shutdown, or its timer
// expires.
//
//
// A scheduler chooses which containers to assign to which idle
// workers, and decides what to do when there are not enough idle
// workers (including shutting down some idle nodes).
//
//
// A syncer updates state to Cancelled when a running container
// process dies without finalizing its entry in the controller
// database. It also calls the worker pool to kill containers that
// have priority=0 while locked or running.
//
//
// A provider proxy wraps a provider with rate-limiting logic. After
// the wrapped provider receives a cloud.RateLimitError, the proxy
// starts returning errors to callers immediately without calling
// through to the wrapped provider.
```

#8 - 08/17/2018 05:31 PM - Tom Clegg

```
// A RateLimitError should be returned by a Provider when the cloud
// service indicates it is rejecting all API calls for some time
// interval.
type RateLimitError interface {
        // Time before which the caller should expect requests to
        // fail.
        EarliestRetry() time.Time
        error
}

// A QuotaError should be returned by a Provider when the cloud
// service indicates the account cannot create more VMs than already
// exist.
type QuotaError interface {
        // If true, don't create more instances until some existing
        // instances are destroyed. If false, don't handle the error
        // as a quota error.
        IsQuotaError() bool
        error
}

type InstanceTags map[string]string
type InstanceID string
type ImageID string

// Instance is implemented by the provider-specific instance types.
type Instance interface {
        // ID returns the provider's instance ID. It must be stable
        // for the life of the instance.
```

```
        ID() InstanceID

        // String typically returns the cloud-provided instance ID.
        String() string

        // Cloud provider's "instance type" ID. Matches a ProviderType
        // in the cluster's InstanceTypes configuration.
        ProviderType() string

        // Get current tags
        Tags() []string

        // Replace tags with the given tags
        SetTags(InstanceTags) error

        // Shut down the node
        Destroy() error

        // SSH server hostname or IP address, or empty string if
        // unknown while instance is booting.
        Address() string


// Return nil if the given public key matches the instance's


// SSH server key. If the provided Dialer is not nil,


// VerifyPublicKey can use it to make outgoing network


// connections from the instance -- e.g., to use the cloud's


// "this instance's metadata" API.

        VerifyPublicKey(ssh.PublicKey, net.Dialer) error
}

type Provider interface {
        // Create a new instance. If supported by the driver, add the
        // provided public key to /root/.ssh/authorized_keys.
        //
        // The returned error should implement RateLimitError and
        // QuotaError where applicable.
        Create(arvados.InstanceType, ImageID, InstanceTags, ssh.PublicKey) (Instance, error)

        // Return all instances, including ones that are booting or
        // shutting down. Optionally, filter out nodes that don't have
        // all of the given InstanceTags (the caller will ignore these
        // anyway).
        Instances(InstanceTags) ([]Instance, error)
}
```

#9 - 08/21/2018 01:56 PM - Tom Clegg

Azure and Amazon have a "run command on instance" API that might let us get the instances' SSH server public keys.

- https://docs.microsoft.com/en-us/rest/api/compute/virtual%20machines%20run%20commands/runcommand
- https://docs.aws.amazon.com/systems-manager/latest/APIReference/API_SendCommand.html

Google doesn't seem to have that, but it does have a related feature, https://cloud.google.com/compute/docs/instances/verifying-instance-identity

- "Your instances must have a service account associated with them so that they can retrieve their identity tokens."
- The dispatcher could connect to the instance's SSH server with host key verification disabled, get the metadata URL using (*ssh.Client)Dial, and validate the returned JWT against Google's public key before doing anything else with the instance.
- User containers would also be able to get a valid JWT, so (at least) we would have to stash the public key in an instance tag before starting any user containers, and use that to verify subsequent connections.

This approach should work on any provider:

When connecting to an instance that hasn't been tagged with its public key / fingerprint, accept any host key, but before doing anything else call a "verify host key" function with the public key and a Dial function that tunnels through SSH. The driver can

- store a nonce in an instance tag, and pull the instance metadata (e.g., http://metadata.google.internal/computeMetadata/v1/instance/tags) through the ssh tunnel, and verify that the nonce appears
- use the cloud's "run command" API to fetch the public key
- use whatever other method makes sense on this cloud

The first approach is not as safe after a user container has run, so it would be best to copy the public key to an instance tag before running any containers, and then use the tag instead of the verification mechanism.

#10 - 08/24/2018 07:37 PM - Tom Clegg

```
// A Driver returns an InstanceSet that uses the given
// driver-dependent configuration parameters.
//
// The supplied id will be of the form "zzzzz-zzzzz-zzzzzzzzzzzzzzz"
// where each z can be any alphanum. The returned InstanceSet must use
// this id to tag long-lived cloud resources that it creates, and must
// assume control of any existing resources that are tagged with the
// same id. Tagging can be accomplished by including the ID in
// resource names, using the cloud provider's tagging feature, or any
// other mechanism. The tags must be visible to another instance of
// the same driver running on a different host.
//
// The returned InstanceSet must ignore existing resources that are
// visible but not tagged with the given id, except that it should log
// a summary of such resources -- only once -- when it starts
// up. Thus, two identically configured providers running on different
// hosts with different ids should log about the existence of each
// other's resources at startup, but will not interfere with each
// other.
//
// Example:
//
//      type provider struct {
//              ownID     string
//              AccessKey string
//      }
//
//      func ExampleDriver(config map[string]interface{}, id InstanceSetID) (InstanceSet, error) {
//              var p provider
//              if err := mapstructure.Decode(config, &p); err != nil {
//                      return nil, err
//              }
//              p.ownID = id
//              return &p, nil
//      }
//
//      var _ = registerCloudDriver("example", ExampleDriver)
type Driver func(config map[string]interface{}, id InstanceSetID) (InstanceSet, error)
```

#11 - 09/04/2018 05:24 PM - Peter Amstutz

How about "InstanceProvider" instead of "InstanceSet"

#12 - 09/04/2018 09:26 PM - Tom Clegg

I'm not super attached to "instance set", but I'm not keen on "provider" because it already means "utility computing provider" in this space, which corresponds to Driver. There's one for AWS, one for GCE, etc.

The idea of InstanceSet is an expanding/shrinking set of instances. A caller can use multiple distinct sets that don't interfere with one another -- whether or not they happen to use the same driver, provider, region, credentials, etc.

We could rephrase the interface so that the Provider/Driver has all the methods, and each method takes an InstanceSetID -- but the dispatcher would need to carry around {Driver, InstanceSetID} tuples anyway, and I think drivers would end up sprinkled with stuff like "is this a new instance set ID that I have to start managing?". It seems easier to make the "start/stop instance set" operations explicit.

#13 - 09/05/2018 03:10 PM - Tom Clegg

- Target version changed from 2018-09-05 Sprint to 2018-09-19 Sprint

#14 - 09/05/2018 03:51 PM - Tom Morris

- Story points set to 1.0

#15 - 09/05/2018 07:57 PM - Peter Amstutz

- Target version deleted (2018-09-19 Sprint)

- Story points deleted (1.0)

InstancePool ? I'm just not keen on the term "Set" because it is usually used to denote a data structure where the key operation is the ability to efficiently determine membership, whereas the key operations of this component are creating, listing, and destroying cloud resources. "CloudResourceGroupManager" sounds enterprisy but would at least be descriptive.

#16 - 09/11/2018 04:07 PM - Peter Amstutz

- Target version set to 2018-09-19 Sprint

- Story points set to 1.0

#17 - 09/11/2018 04:47 PM - Tom Clegg

Hm. I was thinking of a set in the more abstract sense of an unordered collection of distinct items, rather than in terms of the efficiency of a "test for membership" operation.

I'm not keen on "pool" because it connotes a set of initialized objects which can be checked out, used, and then returned to the pool to minimize create/destroy cost. But that behavior is implemented in workerPool, not in the driver. The InstanceSet's add/remove/list operations correspond more directly to the real (and slow) create/destroy/list operations provided by the cloud provider.

#18 - 09/12/2018 01:32 PM - Tom Clegg

13964-dispatch-cloud has a fair bit of this implemented. Remaining todo:

- more test coverage
- finish up loose ends, including:
  - workerpool - probe periodically
  - workerpool - shutdown when idle
  - ssh - public key authentication
  - ssh - host key verification
  - crunch-run - add "--probe" function
  - config/driver glue
- Azure driver
- AWS driver
- GCE driver
- (Optional?) copy crunch-run binary from dispatcher to worker node

#19 - 09/19/2018 03:18 PM - Tom Clegg

- Target version changed from 2018-09-19 Sprint to 2018-10-03 Sprint

#20 - 10/03/2018 03:18 PM - Tom Clegg

- Target version changed from 2018-10-03 Sprint to 2018-10-17 sprint

#21 - 10/10/2018 02:17 PM - Tom Morris

- Related to Story #13908: [Epic] Replace SLURM for cloud job scheduling/dispatching added

#22 - 10/10/2018 03:44 PM - Tom Clegg

- Related to Feature #14325: [crunch-dispatch-cloud] Dispatch containers to cloud VMs directly, without slurm or nodemanager added

#23 - 10/17/2018 01:30 PM - Tom Clegg

- Status changed from In Progress to Resolved

#24 - 11/13/2018 08:51 PM - Tom Morris

- Release set to 14