Volume (and Content) device API

Volume (and Content) device API

We currently implicitly create volumes when deploying application instances and we are adding the support to explicitly create volumes from the controller /UI. The existence of such volumes need to be configured by the controller and reported by EVE to the controller.

For most volumes there is some immutable content (formerly known as images; not called “content trees”) which are used to create the volume. We’d also like to report those in info messages, but the details are less known since it depends on what can be made available from containerd when it comes to layers etc.

However, other volumes will be created from blank space, or merely serve as an adapter to some external storage, in which case there is no associated local content tree.

The kind of config EVE will expect

We are going to see one extra top level EVE config object (to be described in storage.proto) called Volume and we will see our good old object Image (message Image) rename into ContentTree. Volume will be very similar in structure to what used to be known as Drive (buried inside of an app config part of the config). So putting it all together (and marking new/updated part in red):

```
message ContentTree { // previously known as (PKA) “message Image”
  string contentTreeID // UUID
  string dsId // effectively pointer/key into dsConfigs

  string URLsuffix // PKA name - added to the datastore URL
  Format iformat // RAW, QCOW2, ~~CONTAINER~~OCI, BLOB_TREE,

  // the following is only used for individual blobs, if this message references a group of blobs
  // e.g. in the case of OCI -- this information is expected to be provided by a top level blob
  // that this message points to via its URL
  string sha256
  int64 sizeBytes // used for capping resource consumption eve for OCI & BLOB_TREE
  SignatureInfo siginfo
}

message Volume { // previously known as (PKA) “message Drive”
  string volumeID // UUID uuid
  volumeContentOrigin origin
  VolumeAccessProtocols protocols[] // describes all the different ways how this Volume can
                                    // be offered to Tasks (9p, nfs, scsi, ata, virtio, etc.)
  int64 [re]generationCount
  // miscellaneous attributes of the Volume
  int64 maxsizebytes
  bool readonly
 ~~bool preserve~~
}
```

The at the top AppInstanceConfig will change based on these red colors (for transition phases see section at end of document):

```
message AppInstanceConfig {
  UUIDandVersion uuidandversion = 1;
  string displayname = 2;
  VmConfig fixedresources = 3;
  repeated Volume drives = 4; // To be deprecated in phase 3; replaced by VolumeRef
  bool activate = 5;

  ...

  // contains the encrypted userdata
  CipherBlock cipherData = 13;
  repeated string volumeRef = 14; // UUIDs of the volumes
}
```

Types of information to report

Name and identification

An explicitly created volume will have a VolumeID which is a UUID allocated by the controller. A device may rely on the volume ID to be unique across all volumes on a single device, but may not rely upon a device ID being reused on other devices. Depending on how the controller allocates IDs, they might be unique across the infrastructure, or might be unique only across volumes on a single device. However, there is no impact on this API whether or not the controller combines the volumeID with the device UUID.

There is a desire to be able to re-generate the volume from the immutable content. This can be done by creating a new volume with a new ID, but there are use cases where this is cumbersome. Hence it seems useful to add a generationID integer[a][b][c][d][e][f][g][h]; this might be called purgeCounter elsewhere (because the operation is commonly referred to as purging the local modifications.)

As we transition to volumes the controller will explicitly allocate UUIDs for the volumes and include those in the configuration APIs. Thus even the volumes which used to be created implicitly through the Drive in the AppInstanceConfig message will have a volume UUID.

Thus for the purpose of identification we have:

```
// The volume is identifier by volume UUID and a generation number
message volumeName {
  string volumeID = 1;              // UUID string
  int generationCount = 2;
  string displayName = 3;           // Some user-friendly name carried from the UI for debugging?
}
```

Volume Lifecycle

We will reuse the current states which are used for app instances (ZSwState), but states past INSTALLED do not apply to volumes (not that in particular Purging does not apply; a new volume is reported using the new generationCount when there is a purge operation in progress).

Volumes created from blank space will transition from INIT to INSTALLED since there is no download or verification associated with them.

```
message volumeStatus {
  ZSwState state = 1;               // State of Software Image download/install
  uint32 downloadProgress = 2;      // Download progress; 0-100 percent
}
```

Resource Consumption

We at least should have a maxSizeBytes which comes from the configuration.

But it might make sense to have info include the curSizeBytes which is what is currently used from storage for the volume.

TBD: We might want to define a separate metrics message with information about read/write bytes/ops.

```
message volumeResources {
   int maxSizeBytes = 1;              // From config
   int curSizeBytes = 2;              // Current disk usage
}
```

Volume Usage

Knowing when it was created, and a reference count (which could be more than one if shared)

```
message volumeUsage {
   google.protobuf.Timestamp createTime = 1;
   uint refCount = 2;
   google.protobuf.Timestamp lastRefcountChangeTime = 3; // When refCount last changed
}
```

Content Origin

Presumably we will have this to parallel the configuration.

Initially we will need two types: Downloaded Content, and Blank Content. This allows us to add more types for the network storage access without having to pretend that everything is backed by a (local) content tree.

```
enum volumeContentOriginType {
      UNKNOWN = 0;
      BLANK = 1;
      DOWNLOAD = 2;
}

message volumeContentOrigin {
```

```
    volumeContentOriginType type = 1;
    VolumeType voltype // describes the type of the constructed volume (note that "EMPTY" is not used; that is the "BLANK" type)
    string downloadContentStoreID = 2; // where we get DOWNLOAD types from
    // TBD More optional fields for other originTypes
}

~~message volumeDownloadOrigin {~~
~~  string datastoreID = 1;            // UUID string~~
~~  string URLsuffix = 2;              // what to append to the datastore URL~~
~~  string sha = 3;                    // Either specified in config or determined from registry~~
~~}~~
```

Putting it together

```
message ZInfoVolume {
  volumeName name = 1;
  volumeStatus status = 2;
  volumeResources resources = 3;
  volumeUsage usage = 4;
  volumeContentOrigin origin = 5;
}
```

Image/Content information

For the image/content we should extract what we can get from the containerd for the layers. But it is keyed by a hash (and I don't know if we should have a reference to the registry we got it from). The notion of a createTime, refCount, and lastUseTime might make sense for the content.

Biggest TBD is the extent to which we want to represent (and how) the tree of content. Current placeholder is the componentShaList below.

```
message contentName {
  string sha = 1;                  // hash
  string datastoreID = 2;          // UUID string - useful?
}

message contentResources {
   int curSizeBytes = 1;           // Current disk usage
}

message ZInfoContentTree {
   contentName name = 1;
```

```
  volumeStatus status = 2; // Same info as for volumes
  contentResources resources = 3;
  volumeUsage usage = 4;              // Same info as for volumes
  repeated string componentShaList = 5;
}
```

Top-level info message

Following the current scheme we add as in red:

```
enum ZInfoTypes {
  ZiNop = 0;
  ZiDevice = 1;
  // deprecated = 2;
  ZiApp = 3;
  // deprecated = 4;
  // deprecated = 5;
  ZiNetworkInstance = 6;
  ZiVolume = 7;
  ZiContentTree = 8;
}



message ZInfoMsg {
  ZInfoTypes ztype = 1;
  string devId = 2;
  oneof InfoContent {
    ZInfoDevice dinfo = 3;
    ZInfoApp ainfo = 5;
    // deprecated = 10;
    // deprecated = 11;
    ZInfoNetworkInstance niinfo = 12;
    ZInfoVolume vinfo = 13;
    ZInfoContentTree ctinfo = 14;
  }
  google.protobuf.Timestamp atTimeStamp = 6;
}
```

Metrics

TBD but a rough sketch is based on the current diskMetrics with some tweaks to use the volumeName. Note that the used/free semantics depends on the type of volume. For a directory we can report file system usage. For a qcow2 image we can only report how full the qcow2 is relative to its max size.

```
// For Volume; counts since boot

message volumeMetric {

  volumeName name = 1;

  uint64 readBytes = 3;              // In MB

  uint64 writeBytes = 4;       // In MB

  uint64 readCount = 5;              // Number of ops

  uint64 writeCount = 6;       // Number of ops

  uint64 total = 7;              // in MBytes

  uint64 used = 8;               // in MBytes

  uint64 free = 9;               // in MBytes

}
```

Transition plan

As we add support to the controller and EVE we will go through the following steps:

- Today: old EVE, old controller.
- Phase1: old EVE, new controller. Controller is sending both Volume and Drive for the appInstanceConfig.
- Phase2: new EVE, new controller.

The new EVE will upgrade the schema for /persist/img on first boot by using the checkpointed protobuf message from before the reboot.

- Phase 3: new EVE, cleaned up controller. Controller will no longer send Drive in appInstanceConfig; only sending Volume

NOTE If there is a downgrade of EVE during phase2 to an old EVE (which does not support the new schama for /persist/img) the volumes in /persist/img will not be used which can be disruptive for deployed applications.

Considered and rejected ideas

Name and identification using the /persist/img schema

An explicitly created volume will have a VolumeID which is a UUID allocated by the controller. EVE assumes that this UUID is unique across the device on which EVE runs.. However, there is no impact on this API whether or not the controller combines the volumeID with the device UUID.

There is a desire to be able to re-generate the volume from the immutable content. This can be done by creating a new volume with a new ID, but there are use cases where this is cumbersome. Hence it seems useful to add a generationID integer[i][j][k][l][m][n][o][p]; this might be called purgeCounter elsewhere (because the operation is commonly referred to as purging the local modifications.)

Currently the controller implicitly asks EVE to create volumes by the Drive in the API. There are different ways the controller might transition to using volumes for existing, deployed application instances:

1. The controller takes the current Zededa manifest for the application and extracts the drive/image information and uses that to create a Volume object in the controller (with a UUID) and sends that as part of the EVE configuration. Hence even for existing applications there will be explicit volumes with UUIDs.
2. The controller continues to use the Drive message in the API to specify volumes for existing application instances, while new ones use the Volume object. In that case there will be no UUID associated with the volumes implicitly specified by the Drive protobuf message.

If we need to support the second approach in EVE, then we will have volumes which are created implicitly as part of deploying an app instance do not have a volumeID, but can be identified by a combination of the App Instance UUID and the Image UUID (which we might want to rename to “Content Tree UUID”). The content tree in turn might refer to a datastore, have some relative URL/name in that datastore, and any given use of that content tree will have a hash which uniquely identifies it.

Thus for the purpose of identification we have:

```
// If the volume is explicitly created it has a volume UUID

// Otherwise it has a app instance UUID plus a image UUID

// In all cases there is a generation number

message volumeName {

  string volumeID = 1;              // UUID string

  string appInstID = 2;             // UUID string

  string imageID = 3;                  // UUID string = ContentTreeID[q][r]

  int generationCount = 4;

  string displayName = 5;           // Some user-friendly name?

}
```

Note that the appInstID and imageID are only needed if EVE needs to support implicitly created volumes (case 2 above).

[a]I don't understand this. I have some immutable content (image). I generate a volume from it. At that point, an app might or might not change it. If I need a volume that is a fresh, clean version of that volume, I need to generate a new one with a new UUID. How would the generation ID help? I need a new volume.

[b]Well, you have a series of immutable content blobs -- but aside from that you can ask the volumemaneger to basically re-set the Volume to its original state right after the creation.

Think of it this way -- this is getting Volume to the state of the snapshot at the beginning of Volume's life +avi@zededa.com

[c]Understood. But if I am getting it to "the state of the snapshot at the beginning of the Volume's life", then it is identical to the state of a new snapshot, or to discarding all changes since then.

What purpose does the generation ID serve here?

[d]While we are running using generation0 and in the process of downloading, verifying, creating generation1, we want to be able to report the existence of both volumes. Note that we try to minimize the outage for the application to just a reboot using the new generation of the volume.

[e]So the case is:

1. I create an ECO, using volume 111 based on image A, version 1 (A:1)

2. The image for the ECO is updated to A:2. I want to start a new version of the ECO, based on A:2, but I want to keep the ECO around until everything is ready for a near-zero downtime switch

3. I download A:2

4. I create a new volume (111 gen1)

5. I stop ECO, start it on the new volume, and I am good to go.

If that is the case, why make it confusing with gen0, gen1, etc.? Just call it a new volume. The volume UUID is generated by the controller (or by the device, doesn't matter for this scenario). It isn't generated or seen by the end-user.

1. Create an ECO using volume 111 based on A:1

2. Download A:2

3. Create volume 6a4 based on A:2

4. Swap the ECO to run off of 6a4 instead of 111

Both 6a4 and 111 were based on A, different versions, which might just as well be different images; A:1->A:2 vs A:1->B:5 is just an ease-of-use thing. Why confuse it with "generation IDs"?

[f]What part of the API would tell EVE to swap in step 4? The API we have is a purgeCmd counter. We don't have an API to say "replace volume X1 with volume X2 for this app instance".

[g]Actually, I am thinking more "replaced appA complete spec abcd123 with AppA complete spec 543ddf6, and do it rolling".

[h]Well, that isn't what we have in the API today. And I think the notion of updating an app is more natural than replacing. Also, whatever we do I think we need the flexibility to say "update the app container with the new version, but keep the data or empty volume unchanged", as opposed to recreating the empty volume.

[i]I don't understand this. I have some immutable content (image). I generate a volume from it. At that point, an app might or might not change it. If I need a volume that is a fresh, clean version of that volume, I need to generate a new one with a new UUID. How would the generation ID help? I need a new volume.

[j]Well, you have a series of immutable content blobs -- but aside from that you can ask the volumemaneger to basically re-set the Volume to its original state right after the creation.

Think of it this way -- this is getting Volume to the state of the snapshot at the beginning of Volume's life +avi@zededa.com

[k]Understood. But if I am getting it to "the state of the snapshot at the beginning of the Volume's life", then it is identical to the state of a new snapshot, or to discarding all changes since then.

What purpose does the generation ID serve here?

[l]While we are running using generation0 and in the process of downloading, verifying, creating generation1, we want to be able to report the existence of both volumes. Note that we try to minimize the outage for the application to just a reboot using the new generation of the volume.

[m]So the case is:

1. I create an ECO, using volume 111 based on image A, version 1 (A:1)

2. The image for the ECO is updated to A:2. I want to start a new version of the ECO, based on A:2, but I want to keep the ECO around until everything is ready for a near-zero downtime switch

3. I download A:2

4. I create a new volume (111 gen1)

5. I stop ECO, start it on the new volume, and I am good to go.

If that is the case, why make it confusing with gen0, gen1, etc.? Just call it a new volume. The volume UUID is generated by the controller (or by the device, doesn't matter for this scenario). It isn't generated or seen by the end-user.

1. Create an ECO using volume 111 based on A:1

2. Download A:2

3. Create volume 6a4 based on A:2

4. Swap the ECO to run off of 6a4 instead of 111

Both 6a4 and 111 were based on A, different versions, which might just as well be different images; A:1->A:2 vs A:1->B:5 is just an ease-of-use thing. Why confuse it with "generation IDs"?

[n]What part of the API would tell EVE to swap in step 4? The API we have is a purgeCmd counter. We don't have an API to say "replace volume X1 with volume X2 for this app instance".

[o]Actually, I am thinking more "replaced appA complete spec abcd123 with AppA complete spec 543ddf6, and do it rolling".

[p]Well, that isn't what we have in the API today. And I think the notion of updating an app is more natural than replacing. Also, whatever we do I think we need the flexibility to say "update the app container with the new version, but keep the data or empty volume unchanged", as opposed to recreating the empty volume.

[q]Hmm... should this be the image _ID_, or its _hash_?

[r]Currently we refer to all images using a UUID; this is the Image in the Drive in the API. Inside the image there will be a sha.