VM Elasticity Design Document

- Design Goal / Requirement

  To support auto-scaling of the system resources to adapt to the load of external requests to the Brown Dog Data Tiling Service. In general, this includes Medici, MongoDB, RabbitMQ, and the extractors. Currently the design focuses only on auto-scaling the extractors. Specifically, the system needs to start or use more extractors when a certain criterion is met (such as the number of outstanding requests exceeds a certain threshold) – scaling up, and suspend or stop extractors when other criteria are met – scaling down. The purpose of the scaling down part is to save system resources (CPU, memory, etc.) for other uses.

- Investigated technologies
  - Olive (olivearchive.org): mainly developed at Carnegie Mellon University (CMU).

    Three main ideas: Internet Suspend/Resume (allow a user to start using a VM before its entire image is downloaded), indexing and searching of VM images, incrementally composing VM images. The 3rd and part of the 2nd have been done in Docker.

    Impression: seems to be academic quality; the feature and quality do not seem a good fit for the Brown Dog project.

  - OpenVZ (openvz.org): a community project, supported by the company Parallels, Inc. An OS-level virtualization technology based on the Linux kernel.

    Impression: Good for server consolidation, web hosting. Seems to be production quality. For Linux only, so at least does not seem a good fit for the high-level VM architecture technology.

  - Docker (docker.com): automates the deployment of applications inside software containers, providing abstraction and automation of operating system–level virtualization on Linux.

    Impression: Similar to OpenVZ, an OS-level virtualization technology: a container runs the same kernel as the host, so at least does not seem a good fit for the high-level VM architecture technology. OpenVZ functionality + distributed architecture to set up repositories and pull images from and push images to the repositories. OpenVZ is system-centric, while Docker is application-centric, where each container is meant to execute one application. For example, the "docker log <ctid>" command shows the stdout of the command /program running in the container. Popular, under active development.

  - OpenStack (openstack.org).

    As a hardware virtualization technology, OpenStack supports multiple OSes, such as Linux and Windows.

  - Current choice: OpenStack.

    Brown Dog VM elasticity project needs to support multiple OSes, so OpenStack seems a viable high level solution. Currently considering using OpenStack. May consider using Docker on the VMs at a low level if needed.

- Algorithm / Logic

Assumptions:

The following assumptions are made in the design:

1. an extractor is installed as a service on a VM, so when a VM starts, all the extractors that the VM contains as services will start automatically and successfully; if services do not fulfill all requirements, we might have to look into alternatives;
2. the resource limitation of using extractors to process input data is CPU processing, not memory, disk I/O, or network I/O, so the design is only for scaling for CPU usage;
3. the system needs to support multiple OS types, including both Linux and Windows;
4. the system uses RabbitMQ as the messaging technology.

Algorithm:

The VM elasticity module maintains and uses the following data:

1. RabbitMQ queue lengths and the number of consumers for the queues;
   Can be obtained using RabbitMQ management API. The number of consumers can be used to verify that the action to scale up/down succeeded.
2. for each queue, the corresponding extractor name;
   Currently hard coded in the extractor code, so that queue name == extractor name.
3. for a given extractor, the list of running VMs where an instance of the extractor is running, and the list of suspended VMs where it was running;
   Running VM list: can be obtained using RabbitMQ management API, queue --> connections --> IP.
   Suspended VM list: when suspending a VM, update the mapping for the given extractor, remove the entry from the running VM list and add it to the suspended VM list.
   Also maintain the data of mapping from a running/suspended VM to the extractors that it contains. This is useful in the scaling up part.
4. the number of vCPUs of the VMs;
   This info is fixed for a given OpenStack flavor. The flavor must be specified when starting a VM, and this data can be stored at that time.
5. the load averages of the VMs;
   For Linux, can be obtained by executing a command ("uptime" or "cat /proc/loadavg") with ssh. Verified that using ssh connection multiplexing (SSH ControlMaster), we can get it quickly in <1 second, usually 0.3 second. If needed, can use a separate thread to get this data, instead of in-line in the execution flow.

6. for a given extractor type, the list of VM images where the extractor is available, and the service name to start another extractor instance, i.e., a pair of (VM image name, service name). The service name is needed only for running additional extractor instances, since the first instance of that extractor will be started automatically as a service.
   Also maintain the data of mapping from a given VM image to the extractors it contains. This is useful in the scaling up part.
   This is manual and static data. Can be stored in a config file, a MongoDB collection, or using other ways.
7. the last times a request is processed by the VMs, and in the queues.
   The VM part can be obtained using the RabbitMQ management API, /api/channels/: "idle_since" and "peer_host". Need to aggregate the channels that have the same peer_host IP, and skip the ones on the localhost. This info is used in the scaling down part for suspending a VM.
   The queue part can be obtained using the RabbitMQ management API, /api/queues: "idle_since". Used in the scaling down part for stopping extractor instances.

In the above data, items 2, 4 and 6 are static (or near static), the others are dynamic, changing at run time.

Periodically (configurable, such as every minute), the system checks whether we need to scale up, and whether we need to scale down. These 2 checks can be done in parallel, but if so, the system needs to protect and synchronize the shared data, such as the list of running VMs.

Scaling up:

The criterion for the need of scaling up – to start more extractors for a queue – is:

1. the length of the RabbitMQ queue > a pre-defined threshold, such as 100 or 1000, or
2. the number of consumers (extractors) for this queue is below the configured minimum number.

Get the list of running queues, and iterate through them:

1. If the above criterion is reached for a given queue, say, q1, then use the data item 2 above, find the corresponding extractor (e1). Currently this is hardcoded in the extractors, so that queue name == extractor name.
2. Look up e1 to find the corresponding running VM list, say, (vm1, vm2, vm3).
3. Go through the list one by one. If there's an open slot in the VM, meaning its #vCPUs > loadavg + <cpu_buffer_room> (configurable, such as 0.5), for example, vm1 #vCPUs == 2, loadavg = 1.2, then start another instance of e1 on vm1. Finish working on this queue and go back to Step 1 for the next queue. If there's no open slot on vm1, look at the next VM in the list. Finish working on this queue and go back to Step 1 for the next queue if an open slot is found and another instance of e1 is started.
4. If we go through the entire list and there's no open slot, or the list is empty, then look up e1 to find the corresponding suspended VM list, say, (vm4, vm5). If the list is not empty, resume the first VM in the list. If unsuccessful, go to the next VM in the list. After a successful resumption, look up and find the other extractors running in the VM, and set a mark for them so that this scaling logic will skip these other extractors, as resuming this VM would also resume them. Finish working on this queue and go back to Step 1 for the next queue.
5. If the above suspended VM list is empty, then we need to start a new VM to have more e1 instances. Look up e1 to find a VM image that contains it. Start a new VM using that image. Similar to the above step, after this, look up and find the other extractors available in the VM, and set a mark for them so that this scaling logic will skip these other extractors, as starting this VM would also resume them.

At the end of the above iterations, we could consider verifying whether the expected increase in the number of extractors actually occurred or not, and print the result out.

Scaling down:

a. Stop idle extractor instances:
   Find out idle queues (no data / activity for a configurable period of time). For each such queue, find out the running VMs and the number of extractor instances. We allow a user to specify the minimum number of total running instances for an extractor type in the config file. If the number of extractor instances is > 1, stop all instances that still keep the min number of running instances for the extractor type, leaving the first instance running on each machine.

b. Suspend idle VMs.

Get the list of IPs of the running VMs. Iterate through them:

If there is no RabbitMQ activity on a VM in a configurable time period (say, 1 hour), then there is no work for the extractors on it to do, so we can suspend the VM to save resources for other tasks. However, if suspending this VM would decrease the number of running instances for any extractor that runs on it below the minimum number configured for that extractor type, we do not suspend it and will leave it running.

Notes:

1. This logic is suitable for a production environment. For a testing environment or a system that's not busy, this logic could suspend many or even all VMs since there is not much or no request, and lead to a slow start – only the next time the check is done (say, every minute), this module will notice that the number of extractors for the queues are 0 and resume the VMs. We could make it configurable whether or not to maintain at least one extractor running for each queue – it's a balance of potential waste of system resource vs. fast processing startup time.
2. In the future, we could support migrating extractors. For example, if 4 extractors run on vm1, and only extractor 1 has work to do, and there is an open slot to run extractor 1 on vm2 (where extractor 1 is already running), we could migrate the extractor 1 instance from vm1 to vm2, and suspend vm1 – but this is considered lower priority.

- Programming Language and Application Type

The team seems to be familiar, more comfortable with Java, Scala or Python, so we consider using these. This module mainly operates by itself, instead of mainly serving client requests, so it does not have to be a web app. A stand-alone app seems a better fit. Can start with no UI or simple CLI. We plan to start with a stand-alone service implementation, which writes part of the data structure / status into a data storage, such as Redis, Mongo, Postgres, so later another module such as a dashboard, or web app could be developed to display or visualize them, without knowing the details of this module.

- Set up this module as a service.

  As an overall monitoring module in the DTS, this module is important. We plan to use upstart to watch and ensure this module itself is up and running.

- Testing
  - Input
    We plan to use scripts to generate high request rates. We can use different input files/configuration to generate high request rates for a certain set of extractors to test the scaling up part. We can either reduce the request rate, or stop sending the requests to test the scaling down part.
  - Output
    We plan to use the OpenStack CLI tool or web UI (for the VM part), and RabbitMQ CLI tool or web UI (for the extractor part) to verify that the extractors are started, and VMs are started/resumed/suspended as expected.