Programming Assignment 2: Build your own cloud management system using the libvirt API

Back to CS695 home page

Introduction

Many popular cloud management systems and virtualization tools are built over the libvirt API. The libvirt API lets users deploy and manage the lifecycle of VMs spread over multiple physical servers, and works across a variety of backend hypervisors. The goal of this assignment is to familiarize yourself with the libvirt API by using it to build a simple framework to manage cloud applications.

In this assignment, we will build the autoscaling functionality that many cloud management systems provide to the cloud users. Real-life systems and cloud applications today (e.g., web servers, database servers) are built to be scalable, so that system performance can keep up with increasing load. A cloud application deployed on VMs or containers can be scaled either horizontally (by adding more replica VMs) or vertically (by increasing the resources allocated to the existing VMs). We will focus on horizontal scaling in this assignment. The scaling can be triggered in many ways, e.g., when the CPU utilization of the VMs running the application crosses a threshold. Most cloud management systems today come with some autoscaling support. When you run an application on a VM in a cloud and enable autoscaling, the cloud management system monitors the utilization of various hardware resources (CPU, memory, disk, and so on), and spawns a new instance/replica of your application node if some resource utilization crosses a threshold. We will build a simplified version of this feature in this assignment.

Before you begin

Please familiarize yourself with the following components required to solve this assignment.

1. Install virt-manager, libvirtd (libvirt-dev, libvirt-daemon), qemu (qemu, qemuctl, qemu-kvm) and virsh. You can find many helpful links online, like this page. Understand these tools from man pages and/or online links.
2. Use virt-manager to create multiple VMs on your workstation. Make sure your machine is capable of hosting and running these VMs without much performance degradation. Use lightweight OS images for better performance, e.g., you may find it easier to use the lightweight ubuntu server instead of ubuntu desktop. For networking between the VMs, NAT is the default and easiest option. You can give static private IP addresses to the network interfaces in the VMs for ease of use, and the IPs should be in the same subnet as that of the virbr0 interface on your host.
3. Each VM (also known as a domain) has an XML file associated with it, which acts as a config file for that VM. Use virsh commands to see how the XML of an active VM looks like, and try to understand what the different XML objects mean. You can also edit this file to change the resources allocated to the VM. XML config files are useful for creating new VMs with a given configuration.
4. Provision 1 CPU core and 1GB RAM for each VM. At the very least, you will need to run 2 VMs on your physical machine for this assignment. So please make sure your system is good enough to host two VMs comfortably.

5. Familiarize yourself with the various libvirt APIs. You can learn more about libvirt here and here. Once you understand the basics, you can do a deep dive into libvirt APIs. The libvirt API concepts introduces the various APIs. The Application Development Guide is a detailed guide for application developers. The Reference Manual for libvirt is a comprehensive link for all the libvirt modules and APIs.
6. Finally, note that the libvirt APIs are available in multiple programming languages. You may solve this assignment in any language of your choice.

Part A: Autoscaling a client-server application

In the first part of this assignment, you will build a simple **autoscaling client-server application**. You must setup a client that talks to "N" server replicas and sends multiple "requests" to these servers. The servers perform some computation for each request and send suitable responses back to the client. You may use any simple multi-threaded socket based client-server application that you may have developed in previous courses (e.g., a simple key-value store), or you may choose to use a more realistic application (e.g., a web client and server). The choice of the application, and which requests/responses to handle, are completely left to you. We are aiming to do autoscaling by monitoring the CPU utilization of server VMs, so it is best if you pick an application where the request processing is CPU-intensive.

You must design your server application to be horizontally scalable. That is, when requests get distributed across replicas, you must ensure that a server replica is capable of handling all requests coming to it. For example, if you are building a key value store, you must ensure that a server replica is able to handle all get/put requests coming to it, say, by sharing the key-value database across server replicas. You may make any simplifying assumptions as required to enable easy sharing of distributed state across replicas. Or, to make your life simple, you can use a stateless application server, where the server performs some CPU-intensive computation on the request to generate a response, without requiring any stored state to generate the response.

Your client must be capable of sending multiple concurrent requests to multiple server replicas in parallel, in order to fully load the servers. That is, your client should be able to saturate the CPUs of the N servers without becoming a bottleneck itself. You may use a multi-threaded architecture at your client to efficiently generate load. The client can use any policy to distribute requests to servers, e.g., round robin, or divert specific requests to specific servers using some intelligent logic. Further, your client must be capable of generating variable amounts of traffic. For example, in a "low load" mode, your client must generate a low amount of traffic such that none of the N servers are overloaded. In a "high load" mode, the client must be able to saturate the CPUs of all N servers. The client can shift between these modes at fixed time intervals, or on receiving external triggers. The client can run directly on the host or inside a VM.

The crux of this assignment is building a monitoring/autoscaling program. This program monitors the CPU utilization of all server VMs in order to detect overload and perform autoscaling. For this part of the assignment, we are expecting you to demonstrate autoscaling in the following manner. Start your server application on N replicas, and your client in a low load mode. The monitoring program must monitor the CPU utilizations at the server replica VMs. When the client shifts to a high load mode, the monitoring program should detect overload, must spawn a new server replica VM, and notify the client. The client must then start using the N+1 servers to serve requests, which should hopefully ameliorate the overload situation at the existing servers.

You are free to decide the communication/notification mechanism between the autoscaling program and your client, in order to let the client know that a new server replica is available. You can also implement the autoscaling program in any programming language of your choice that has support for the libvirt APIs. The only constraint we impose on the monitoring and autoscaling program is that it should **perform the monitoring and autoscaling by invoking the libvirt APIs**, and not by any other means (e.g., use the `system` command to run other commandline tools).

You must choose the load levels in the low and high modes of your client carefully. If your high load is much larger than your low load, the autoscaling application may continue to send out overload triggers even after spawning a new replica. For the sake of simplicity, you may want to make your high load level just slightly higher than your low load level, so that the overload can be mitigated by spawning just one extra server replica.

In this assignment, we are expecting you to demonstrate a **simple scale up from N to N+1 servers, where N=1**. That is, you must start your application with 1 server. When under overload, your autoscaling program should spawn the second server, which should mitigate the overload situation. We will not be evaluating your assignment for higher values of N, though you are welcome to build a generic autoscaling framework that works for all values of N, subject to the constraints of your system to host VMs.

You are also responsible for designing a "demo" to showcase your autoscaling logic. You must think about you will convince the instructor/TA that your code works correctly. You may want to plot some metrics before and after the autoscaling event to demonstrate that your solution has worked. For example, you can measure the CPU utilizations of the server replicas, and show that the overload has reduced. You may also measure the average throughput or latency of request processing at the servers, and show that the performance has improved after autoscaling.

Some helpful hints:

- When the autoscaling program spawns a new server VM, the server application in the VM should start as soon as the VMs boot up. You may also want to start the server as a background process (check out what happens if you start the server as a foreground process). You can modify the rc.local or .profile or init.d files to achieve this goal. Some helpful links are here and here.
- You will be mainly needing the libvirt-domain and the libvirt-host APIs for writing the monitoring program in this assignment. Of course, you are free to use any of the other libvirt APIs as well. The first thing you should do in your monitoring program is to get a `virConnectPtr` handle, which represents a connection to the hypervisor (see Chapter 3. Connections). You need to then get hold of the `virDomainPtr` object for performing various actions like CPU utilization monitoring on the domain. Both `libvirt-domain` and `libvirt-host` modules provide APIs for determining how much resource a particular VM is using.
- This link explains how you can calculate the CPU utilization of a domain. An interesting observation is that if you check out the CPU utilization from inside the server VM (say, using `top`), it will be less than the value which your monitoring program reports. Try to find out which process(es) is/are responsible for this discrepancy.
- You may assume that the original N servers are already up and running before your monitoring program starts. Your autoscaler only needs to worry about spawning new VMs to handle overload. You may use the "create domain" and "define XML" APIs in the `libvirt-domain` module to spawn new VMs (other options exist too).
- Your monitoring program can use any heuristic of your choice to infer an overload situation. For example, you can check that the CPU utilizations of all N servers are above a threshold for a sufficiently large period of time. You must be careful in avoiding false positives and triggering autoscaling for transient CPU spikes.

Part B: New features and extensions (ungraded, optional)

You can think of your monitoring and autoscaling application as a very simple cloud management system that is managing the various VMs of an application. You can extend this assignment in many ways to make it more realistic. Below are a few suggestions:

- Build a generalized autoscaler that elastically scales the server application in response to varying load. For example, the elastic scaling logic should increase the number of server replicas to handle overload, as well as shut down some replicas when the system is underloaded. The elastic scaling logic should ensure that you have just enough replicas to handle the incoming traffic, not more and not less.
- You can extend the autoscaling logic to account for server failures. The monitoring program can notify a client when a server fails, so that the client can avoid sending traffic to the failed replica. Your monitoring program can spawn a new server replica to replace the dead one, in order to reduce the impact of failures on the application user.
- You can extend your autoscaling logic to spawn more than one physical machine. Note that libvirt allows you to manage VMs on remote machines as well. If you run out of resources on the existing physical machine, you can spawn new VMs on other physical machines, or even live migrate some VMs to other machines. Similarly, when scaling down the number of replicas, you can consolidate VMs over a small number of physical machines to save resources.
- You can build a GUI-based cloud management interface using the libvirt API. The GUI can take inputs on how many VMs to spawn, which images/applications to run within the VMs and so on. The GUI can also display real time metrics like CPU utilization from the various VMs, and alert the user to events like failures or autoscaling.
- Feel free to play around with existing cloud management systems to come up with your own cool idea!

Submission and grading

You must solve this assignment individually. You must only submit code for part A. Create a tar-gzipped file of your submission, with the filename being your roll number, and upload it on Moodle. Grading this assignment will involve a demo and viva with the instructor/TA.

Good luck!

Back to CS695 home page