VIRTUALIZATION AND DEPLOYMENT MANAGEMENT FOR THE KAT-7 / MeerKAT CONTROL AND MONITORING SYSTEM

Neilen Marais, SKA, Cape Town, South Africa

Abstract

To facilitate efficient deployment and management of the Control and Monitoring software of the South African 7-dish Karoo Array Telescope (KAT-7) and the forthcoming Square Kilometer Array (SKA) precursor, the 64-dish MeerKAT Telescope, server virtualization and automated deployment using a host configuration database is used. The advantages of virtualization are well known; adding automated deployment from a configuration database, additional advantages accrue: server configuration becomes deterministic, development and deployment environments match more closely, system configuration can easily be version controlled and systems can easily be rebuilt when hardware fails. We chose the Debian GNU/Linux based Proxmox VE hypervisor using the OpenVZ single kernel container virtualization method, along with Fabric (a Python SSH automation library) based deployment automation and a custom configuration database. This paper presents the rationale behind these choices, our current implementation and our experience with it, and a performance evaluation of OpenVZ and KVM. Tests include a comparison of application specific networking performance over 10GbE using several network configurations.

INTRODUCTION

The Control and Monitoring (CAM) subsystem of the current KAT-7 and under-construction MeerKAT telescopes [1] mostly consists of high-level control software written in Python, talking KATCP (Karoo Array Telescope Control Protocol) [2] over Ethernet to lower level controllers. While the CAM software is logically fairly complex, pulling together a large number of distributed lower-level controllers into a single cohesive telescope, the use of an Ethernet fieldbus allows the core CAM to be hosted on a physically centralised set of servers.

Around the end of 2011, KAT-7's history as an experimental prototype was showing. Work was undertaken to improve the deployment management from a point where CAM software was hosted directly on servers that were maintained by hand. Deployment of a new CAM software revision was a fraught process, and hardware failure resulted in extended downtime. Only a limited number of development environments, none which matched the deployed environment closely, was available. We set out to: have deterministic and repeatable system configurations; keep versioned configuration history by storing configuration scripts under version control; minimize the number of manual steps required for deployment; minimize downtime when deploying software; allow quick revision roll-back; minimize downtime in case of CAM system hardware failure; provide better isolation in terms of resource usage between components that share a physical server; quickly and easily deploy a number of development environments on a limited development hardware resource; and deploy development testing environments that match deployment environments quite closely.

CHOSEN TECHNOLOGIES

Proxmox VE as Hypervisor

Several server virtualization technologies are available; they may be graded between the extremes of Full Virtualization where a complete virtual computer is emulated and Virtual Private Servers where a single hypervisor OS kernel is shared by multiple *containers*; the kernel has accounting and isolation features to make each container act like a separate server without the overhead of hardware emulation or multiple OS kernels, and no special hardware support is needed for good performance.

The Debian GNU/Linux based Proxmox VE [3] hypervisor supports both kinds of virtualization. The majority of the CAM system runs on a common GNU/Linux platform and is deployed to containers. Full virtualization is available if e.g. an MS Windows server needs to be deployed. Other desirable features are: our familiarity with Debian-like systems; positive prior experience by our IT department; new containers can be provisioned, stopped and started quickly; ability to perform zero-downtime backup snapshots; the simplicity and speed of installing the base Proxmox hypervisor on a fresh server; the ease of scripting container management over SSH; Free and Open Source (FOSS) licensing; an easy to use Web UI for simple server management and diagnostics. Other advanced Proxmox functionality, such as live container migration and high availability clusters, is not currently utilized.

Virtualbox for Workstation Based Virtualization

Developers and commissioners use a mix of Linux, Apple and MS Windows workstations. To provide a common local development environment, Toy-KAT VMs that run the full CAM software stack but simulate only a subset of the telescope are used. Virtualbox, a FOSS virtualization environment that works across multiple platforms, was chosen for hosting Toy-KAT VMs. Virtualbox is not part of our production deployments, but experiences with it as a developer tool have been quite positive.

ISBN 978-3-95450-139-7

Fabric as Deployment Automation Tool

Fabric [4] is a Python SSH automation library allowing scripted operations defined in Python code to be performed on a remote machine over SSH. Script logic is defined in Python, not in typical Unix shell script. Fabric can also be used to script non-Unix hosts that have an SSH interface. It requires no central management server; Fabric routines can be run from any machine that has SSH access to the nodes being managed. Configuration management systems (e.g. Puppet, Saltstack) were investigated, but seemed to require a significant upfront time investment.

The CAM Fabric library is architected around the concept of host roles; each unit of functionality is described as a role. Each role defines a list of OS level software dependencies (installed using apt-get), a list of Python library dependencies (installed from the Python Package Index (PYPI) or the SKA SA Subversion repository), and optionally a Fabric task script to perform steps like configuring NFS mount points, configuring cron jobs, initializing database schemas and, common to most nodes, checking out, building and installing the appropriate version of the CAM software. The Fabric scripts do not have hard-coded information about CAM nodes; assignments of roles to nodes are read from the host database described below, and are dispatched to appropriate role-handling functions.

Custom Configuration Database

The host configuration database is stored in an INI format file with specified semantics. Each logical node has a section keyed by host-name. This section contains all the information required to deploy a node, including: network configuration (IPs, gateways, etc.), CAM configuration settings, host roles and whether this is a production node, physical hosting information, such as whether it is a virtualized node or not, and for virtualized nodes: name of the host server, hardware resources (i.e. CPUs/RAM, etc.), and an organization-wide unique container ID (CTID) number. A unique CTID is required for each container/VM on a single Proxmox host; having unique CTIDs allows containers to be moved to arbitrary Proxmox hosts at any time.

Related nodes can be placed into one or more groups, allowing operations over a group to be specified with a single command. A Python library module that parses the host file and makes it available as a queryable data structure has been created. This module is integrated with the CAM Fabric library, allowing a Fabric script to access information about the node that it is operating on, and allowing Fabric operations on groups of nodes to be specified on the basis of configuration database queries.

Performance

Performance and efficiency was one of the reasons why container-based virtualization was chosen. Initial lab testing showed that virtualized performance was adequate and, indeed, aggregate CAM server resource utilisation in production was little changed by the adoption of virtualization. However, the soft-realtime [1] design of the CAM subsystem makes it fairly insensitive to server performance, provided enough aggregate CPU throughput is available.

To test the performance of different virtualization approaches, the high speed multi-core SPEAD [5, 6] data-capturing software used by the SKA SA Science Processing subsystem to capture astronomical data produced by the telescope was benchmarked. This test involves capturing a high-speed UDP data-stream over a 10 GbE interface without packet loss and assembling it into SPEAD data-heaps for further processing, stressing the network IO, memory and CPU performance of a system.

The system under test (SUT) is a SUN FIRE X4150 with 2x Intel(R) Xeon(R) E5450 CPUs, 16 GB RAM and a Gen 1 Myricom Myri10GE 10GbE adaptor, a relatively old system with about 50% the per-core performance of modern Xeon CPUs. It does support Intel VT-x virtualization acceleration. Originally procured as a data-capture machine for the Fringe Finder development telescope, low level tests performed at that time showed that this combination of hardware is limited to a raw receive rate of about 6-7 Gb/s, even using jumbo frames. A modern Dell R720 server with a similar Myricom Myri10GE 10 GbE adaptor, capable of transmitting the test SPEAD data-stream at up to 9.7 Gb/s, is the sender. The SUT and sender were connected using a Fujitsu XG2000c switch over copper CX-4 interconnects.

Each test consisted of three runs, sending the same 5 GB SPEAD stream with 654817 SPEAD heaps to the SUT using the `speadtx` command. The `speadrx` command was run on the SUT, confirming the number of packets received; for each configuration the send rate was adjusted down until the SUT could capture all the packets without loss three times in a row. The receive rate is averaged over the three consecutive receive runs. CPU utilization was calculated over a 10 second CPU usage measurement on the host OS while receiving the SPEAD stream. The host measurement takes virtualization CPU overhead into account.

The baseline performance was measured using a direct install of Ubuntu Linux 10.04 LTS 64-bit (kernel 2.6.32-21.32-server), our current production environment, onto the SUT. Jumbo frames (MTU 9000) were enabled allowing each SPEAD heap of the test data to be transmitted as a single UDP packet. Furthermore, the kernel network tuning parameters in the `notes` file in [6] were used. These two optimisations improved the receive rate from around 300 mb/s to the measured 5.5 Gb/s. Using 6 `speadrx` processes gave the best results for the 8-core SUT. Other performance optimisations were attempted, including the use of the newest vendor provided NIC drivers , setting network irq handling CPU affinity to specific CPUs, and tuning the value of `rx-usecs` and `adaptive-rx` (irq-coalescing settings) using the `ethtool` command. It was also confirmed that write combining and MSI interrupts were enabled. These changes made no significant difference to receive performance.

After the baseline measurements were made, the SUT was re-installed with the Proxmox VE hypervisor, version

ISBN 978-3-95450-139-7

Table 1: Virtualized Network Performance

Config	Rate (Gb/s)	CPU use (%)	relative rate (%)	CPU per Gb/s (%)
Baseline	5.49	65.8	100.0	12.0
Host	4.80	59.2	87.5	12.3
OVZ excl	4.65	61.2	84.7	13.2
OVZ veth	3.72	21.1	67.8	5.8
OVC venet	3.86	20.3	70.4	5.3
KVM virtio	2.39	60.5	43.6	25.3

3.1, running the Linux 2.6.32-23-pve-109 kernel which is based on RHEL6. A performance tuning process, as for the baseline, was performed for each of five configurations that were benchmarked. In each case it was necessary to set the MTU on all of the virtual and physical network interfaces and bridges involved. For the KVM VM the kernel tuning must also be done on the VM kernel. The configurations are:

Host Unvirtualized Proxmox VE hypervisor host OS segment. This is a Debian 6.0 64-bit system running a specialised Linux kernel and hypervisor utilities.

OVZ excl OpenVZ container to which the hardware 10 GbE device is assigned with exclusive access.

OVZ veth OpenVZ container with a virtual layer-2 Ethernet device connected via a software layer-2 bridge.

OVZ venet OpenVZ container with virtual layer-3 IP network device connected a software bridge.

KVM virtio Linux Kernel-based VM using a paravirtualized Ethernet device connected to via a software bridge, running Ubuntu 12.04 LTS.

All containers were based on the Ubuntu 10.04 LTS 64-bit template. The `speadrx` binary from the baseline test was used for all tests. Ubuntu 12.04 LTS was used in the KVM VM, since 10.04's virtio driver had major performance issues; it was unable to receive at more than about 200 mbit/s, even slower than using an emulated hardware Ethernet device. KVM with exclusive network hardware access could not be tested, since the Blackford 5000P chipset in the SUT does not support Intel VT-d IO MMU virtualization. All virtualized environments were assigned 8 CPU cores and 12 GB RAM; the test used less than a GB of RAM. No other containers or VMs were active on the Proxmox host when tests were run. Maximum throughput was always achieved at well below 100% CPU utilization, and no single core was fully utilized. A possible explanation is an interaction between kernel packet processing latency and hardware buffer sizes. The unvirtualized Proxmox Host result is ∼13% slower than the baseline, although it uses only 2% more CPU per unit throughput; that may caused by extra overhead introduced by the OpenVZ patches, but that particular kernel may contain additional modifications relative to the Ubuntu 10.04 LTS kernel.

The OpenVZ excl config only incurs a further penalty of 3%, making it a good option when network performance is critical but the management and isolation advantages of a container are desired. The OpenVZ documentation states that venet networking is faster and lower-overhead than the veth; while this is strictly true, the absolute difference is fairly small. Both venet and veth networking are ∼30% slower than the baseline, or ∼18% slower than the unvirtualized Proxmox host. Surprisingly, the CPU usage per unit of throughput is lower for both veth and venet than for the baseline, or indeed, any other configuration. The KVM virtio throughput is less than half that of the baseline, while using more than twice as much CPU per unit of throughput.

While specific benchmarks might be required for high speed data capturing applications, it is clear that, even using older server hardware, any of the tested virtualization configurations would comfortably saturate the 2x1 Gbit Ethernet ports as planned for MeerKAT CAM[1], validating the choice of OpenVZ containers for the CAM subsystem from a performance perspective.

DEPLOYMENT IN PRACTICE

Physical Deployment

The operational CAM subsystem is physically hosted in two geographic areas separated by about 700 km: the telescope site in the sparsely populated Karoo desert to avoid radio frequency interference (RFI), and the operations and engineering site close to Cape Town. The site is connected to the Cape Town office via the high speed SANReN fibre network.

The KAT-7 CAM subsystem servers (and equipment from other subsystems) are hosted in the Compute Container (CC) at the Karoo site: a cooled, RF shielded ISO container with several 19" equipment racks. The CAM servers consist of 2x Dell R410 dual-socket servers and a shared NAS for persistent data. The MeerKAT equipment will be housed in the RFI shielded underground Karoo Array Processor Building (KAPB) currently under construction. The CAM development environments are hosted on set of similarly specced servers and some older Dell and Sun PC servers hosted at the Cape Town office.

The servers at both sites are meant to be generic and easily replaceable. An additional server will be deployed to the Karoo to serve as a KAT-7 cold-spare, and the MeerKAT design also calls for cold-spares. The servers all run the Proxmox VE hypervisor, allowing the rapid deployment of arbitrary logical functions to any server.

The CAM operator workstations (27" iMacs) and display servers (generic PCs) are located in a dedicated control room at the Cape Town offices. Secondary operator workstations are hosted in the Control and Monitoring Container (CMC), another shielded ISO container on-site, and the Meysdam Karoo Operator Center about 50km from the core site.

Logical Deployment

The functions of the CAM systems are deployed to logical nodes; CAM logical nodes are hosted as OpenVZ con-

ISBN 978-3-95450-139-7

tainers on the physical hardware described above. Operator workstations are mostly generic, requiring only basic software such as web browsers and SSH clients. CAM functions also depend on central network services such as a Subversion server managed by the IT dept in Cape Town.

The logical architecture is fully defined by entries in the configuration database. Deployment involves two steps: provisioning of nodes, and node configuration. Nodes are provisioned by creating and assigning hardware resources to a container on a host server, and configuring its networking using a Fabric command such as: `fab proxmox.create_containers_by_group:karoo_system_nodes,700`. This single command will provision all the containers of the production telescope by running commands on the appropriate Proxmox hosts using information in the configuration database. The `700` is a version prefix added to the CTID stored in the database, allowing multiple versions of a deployment to co-exist; versions are switched by stopping and starting the appropriate container versions. After a deployment old containers are kept until the quality of the most recent revision has been proven.

Node configuration involves running software installation and configuration scripts on the virtual nodes, using a command such as: `fab kat_deploy.install_nodes_by_group:karoo_system_nodes,karoocamv7-requirements.txt`; The requirement files are version controlled along with the rest of the CAM software. The specific actions performed on each node are determined by the roles assigned in the configuration database. The `karoocamv7-requirements.txt` indicates the file from which the software requirements should be read, defining the versions of all CAM and external Python packages to be installed.

Persistent Data

Persistent production data is stored on a SAN device hosted in the Karoo CC, shared with the Science Processing subsystem, and is exposed as NFS exports. Persistent CAM data includes: Postgres telescope management databases, sensor data archives, component log files and observation logs. Containers mount the appropriate NFS exports for their roles. While the containers are disposable (can be re-built in ∼10 minutes), the persistent data is mirrored to the Cape Town site every hour. Inter-version compatibility is affected by changes in database schema, but the CAM schema has been fairly stable over the last two years.

DEVELOPMENT ENVIRONMENT

Development is done on fully simulated systems (i.e. hardware devices are replaced by software simulators) and hardware integration on a lab system where some simulators are replaced with representative hardware. Primary development proceeds on workstation-hosted Toy-KAT systems. Running on a single virtual node, aspects of the complete telescope cannot be tested on a Toy-KAT; hence each developer has a set of development containers. They are included in the configuration database and can be created and destroyed by the owning developer using the same Fabric scripts as for the production nodes. Furthermore there are two shared virtual production environments (VKaroo and DevKaroo) where production network and node configurations (down to IP addresses) are emulated using virtualized networking, allowing complete production deployment processes and configurations to be tested. VKaroo runs the current production software revision, while DevKaroo runs the most recent development version.

Adopting virtualization has allowed cost saving by allowing over-subscription of development servers. Previously, a physical server had to be dedicated to each node of a development system. Since each hardware node has enough CPU power to run several development environments simultaneously and since most development environments are only used for short periods of time, dozens of environments can be hosted on four development servers.

EXPERIENCES AND CONCLUSION

While reliable deployment on-site is the main reason why virtualization and automated deployment were originally considered, the most regular advantage is the ability to easily deploy realistic development/testing environments, and quickly switch between software versions by switching containers. We have found the need to add confirmation steps for operating on production nodes and to manage deployment steps depending on external internet services by using local mirrors (PyPi and Ubuntu repositories).

The goals set out in the introduction have largely been met. Future work includes: deployment to fresh containers in the Continuous Integration process and running full scale integration / functional tests (that take hours to complete) on them nightly, and automatic daily building of Toy-KAT VMs. Work on the MeerKAT deployment should process should largely see more of the same, although the network configuration will be more complex [1].

REFERENCES

[1] L. van den Heever, “MeerKAT Control and Monitoring - Design Concepts and Status”, SKA SA, October 2013, ICALEPCS 2013 Proceedings.

[2] S. Cross et al., “Guidelines for Communication with Devices”, SKA SA, July 2012, `http://pythonhosted.org/katcp/_downloads/NRF-KAT7-6.0-IFCE-002-Rev5.pdf`

[3] Proxmox Server Solutions GmbH, “Proxmox VE”, `http://pve.proxmox.com`

[4] C.V. Hansen and J.E. Forcier, “Fabric”, `http://docs.fabfile.org/`

[5] J. Manley, et al., “SPEAD: Streaming Protocol for Exchanging Astronomical Data”, CASPER, June 2012, `https://casper.berkeley.edu/wiki/SPEAD`

[6] A. Barta, “SPEAD”, `https://github.com/ska-sa/spead`

ISBN 978-3-95450-139-7