Testing and Improving Deployment of ATLAS Releases to CVMFS

Tomáš Stefan

October 1, 2018

1 Introduction

My task at CERN was to extensively benchmark CernVM File System (CVMFS), especially a new functionality allowing simultaneous publications that did not have a wide adoption yet.

For these purposes four Athena (ATLAS software) releases were chosen, providing data big enough to be a realistic scenario of files published into CVMFS, specifically:

- Athena_22.0.1_x86_64-centos7-gcc62-opt
  master/x86_64-centos7-gcc62-opt/2018-07-04T2126
- Athena_22.0.1_x86_64-slc6-gcc62-opt
  master/x86_64-slc6-gcc62-opt/2018-07-04T2055
- Athena_21.9.0_x86_64-slc6-gcc62-opt
  21.9/x86_64-slc6-gcc62-opt/2018-07-04T2200
- Athena_21.0.75_x86_64-slc6-gcc62-opt
  21.0/x86_64-slc6-gcc62-opt/2018-07-04T2208

I was given access to five physical machines and four virtual machines. Specification for the **physical machines**:

`RAM`	64 GB
`CPU(s)`	32 (16 cores × 2 threads)
	Model Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz
`HDD`	1 TB
	Model SAMSUNG MZ7KM960 Rev: 003Q

and for the **virtual machines**:

`instance`	m2.2xlarge
`RAM`	32 GB
`vcores`	16
`technology`	KVM

2 Work overview

For an easier and less error-prone solution Ansible was chosen – this is a tool for automation. Ansible connects to the machines through classic SSH, which means there is no need for an additional client running on the destination system.

The configuration is written in YAML format and the file is called a playbook. The playbook is composed of plays, which is a list of tasks for a specified set of machines. In my case I have two groups of machines called *cvmfs-gw* and *cvmfs-rm* and they both have a parent group *cvmfs*. It means when I write a play for host *cvmfs*, it will run the specified tasks in the same order on all machines from group *cvmfs-gw* (effectively just one) and also on all machines from group *cvmfs-rm*.

The common setup of a specific functionality can be placed in `roles`. I am using several of them:

`cvmfs_server`
: build and installation from source code of specific `cvmfs_server` version specified by Git commit hash. The installation is triggered only when there is no `cvmfs_server` yet or the current version doesn't match string from configuration. The string for version matching is in variable *cvmfs_version_check_text* and commit hash in *cvmfs_version_install_hash*.

`cvmfs_gateway`
: install `cvmfs-gateway` package from a repository, version is defined by *cvmfs_gw_version* variable

`athena_local_repository`
: download the Athena releases specified by *athena_releases* variable and all the dependent RPMs and create a local YUM repository. This is especially useful for repeated tests, but has also huge benefit – getting rid of the fluctuation in network download time.

`ayum_installer`
: download AYUM installer utility and add to it the local repository created by `athena_local_repository`

`cvmfs_local_empty_repo`
: create a CVMFS repository with name specified by *cvmfs_repo* variable. If the repository already exists, delete it first. If this is run on the gateway, also add the gateway key in /etc/cvmfs/keys/-{{ cvmfs_repo }}.gw.

`cvmfs_release_manager`
: setup the release managers – add remote repository from the CVMFS gateway. This needs an SSH connection from all release managers to the gateway machine without a password (preferably with a key).

Another nice feature is `tags`. You don't need to always run the whole playbook, instead you can divide it into logical parts, give them an unique tag and then run just the parts you need. That is especially beneficial with huge playbooks running several hours. The tags set in my case:

`cvmfs_server`
: run the `cvmfs_server` role on all machines inside of the *cvmfs* group

`cvmfs_gateway`
: run the `cvmfs_gateway` role on the gateway machine

`athena_local_repository`
: run the `athena_local_repository` role on all release manager machines

`setup`
: the three previous tags together

`install_to_local_disk`
: measured installation (AYUM) of all the releases, one by one, to a local disk – no CVMFS involved. Run on all RMs.

`install_to_local_cvmfs`
: measured installation of all the releases to a CVMFS repository on the same machine. The repository is recreated before each release installation starts. Run on all RMs.

`simultaneous_{1..4}RM`
: first setup the CVMFS GW and RM. Then install simultaneously first specified Athena release from the first RM, second release from the second RM, etc., to the gateway.

`tarball_ingestion`
: needs CVMFS repository with all the Athena releases installed in it. First it archives (tar) those releases to a local disk (if the tar files are not already present). Then it measures the tarball ingestion of all the releases to a local CVMFS repository. If the releases

needs to be tested afterwards, the variable *keep_ingested_releases* needs to be set to `True`.

`tests`
: run all the tests

`clean_all`
: remove everything produced during the setup and tests. Backup the `results` files from *working_dir* on each machine from *cvmfs-rm* group first before running this tag. This is not run unless explicitelly trigerred.

Example of running the playbook with only the ingestion part and variable set for keeping the data for future testing:

```
$ ansible-playbook playbook.yml
      --tags tarball_ingestion
      --extra-vars "keep_ingested_releases=True"
```

2.1 Results

All the results together with the Ansible playbook (and other files) can be found on `https://gitlab.stdin.cz/ts/bench-athena-install`.

The first comparison in Figure 1 is between physical and virtual machines. The installation is done using the AYUM command and the destination is a local disk on the same machine. The result times on the physical machines are very consistent, all of them finished in span of 5 seconds and the avarage value is 812 s. On the other hand, virtual machines have a difference between the fastest and the slowest of 160 s and are noticeably slower with the avarage value of 1175 s.

The Figure 2 shows performance of a local CVMFS repository – installation and publication to the CVMFS repository on the same machine. The installation time alone is comparable to the time of local disk installation from Figure 1. The main difference is the additional time for the publication into CVMFS that is on average 144 s for the physical machines and 233 s for the virtual machines.

Figure 3 compares publication time with ingestion time. Ingestion is an alternative, relatively new way, to get data inside of a CVMFS repository. The input is a tar archive that will be unpacked inside, with the benefit that there is no need for starting a new transaction and publication after the changes, all of that is done internally. However, the disadvantages are

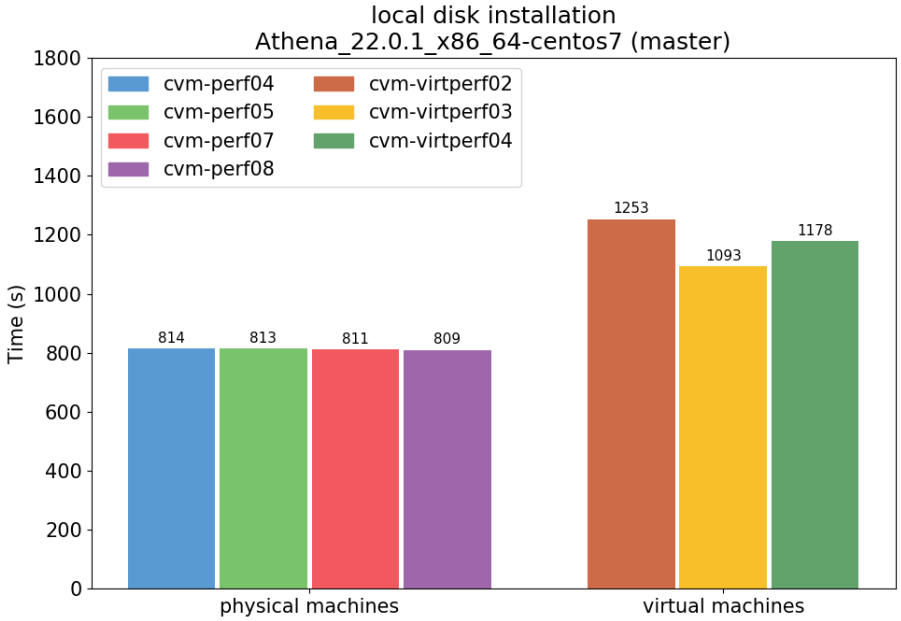


Figure 1: Physical vs. virtual machines – local disk

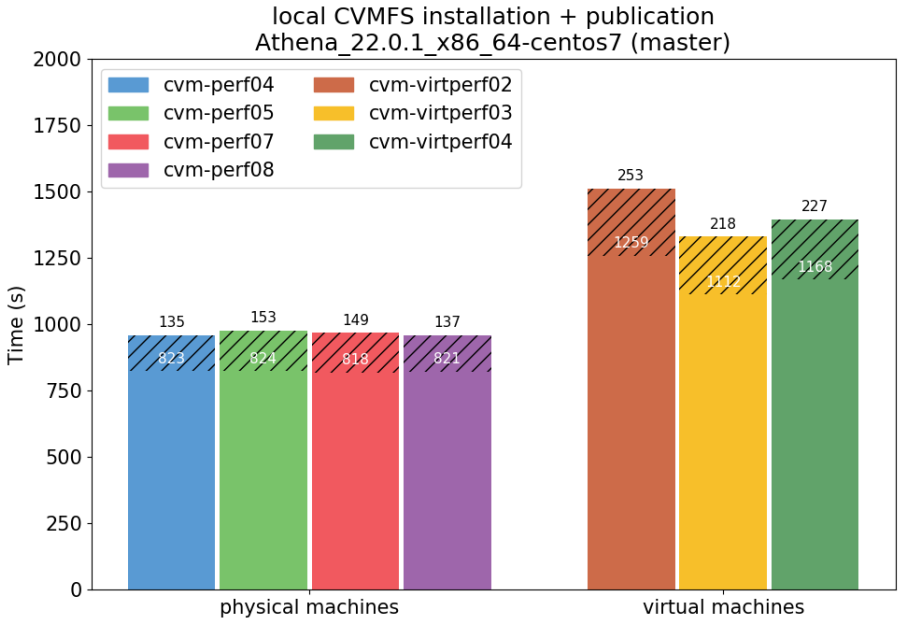


Figure 2: Physical vs. virtual machines – local CVMFS

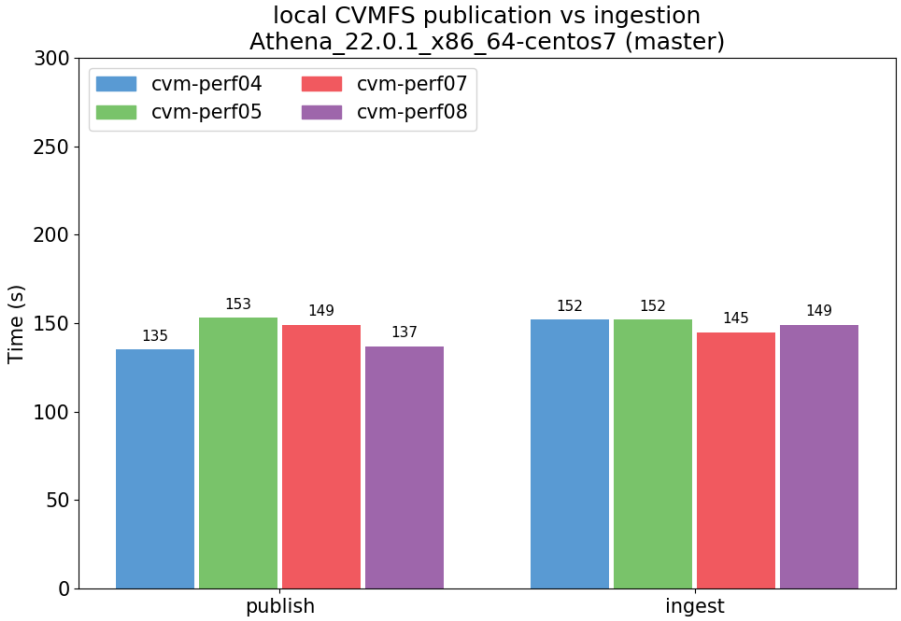


Figure 3: CVMFS publication vs. ingestion

the need to create the archive in advance and the inability to run a post-installation script (only after starting a new transaction). In this case, the average time is 144 and 150 s for publication and ingestion, respectively.

And the last and most complex plot is in Figure 4. The main goal here was to compare overhead with increasing number of simultaneous publications into CVMFS repository. In the past, only one publication at a time was possible (locking the whole repository). However, new features were added and currently it is also possible to have multiple CVMFS release managers for one CVMFS gateway and do several publications at the same time if the destination paths are different.

The group of four grey bars on the left side contain reference values – results of installation to local CVMFS for each of the four chosen Athena releases. And the following four groups of bars are simultaneous installations from n CVMFS release managers to the CVMFS gateway, where n is in order 1, 2, 3, and 4. The results from this run are in Table 1, where n is the number of simultaneous publications started at the same time, `Ideal` is the value of the highest bar from reference values from the first n bars, `Current` is the sum of values from first n bars from reference values, and `Simultaneous` is the value of highest bar from group of n simultaneous events.

During all tests with a target of a CVMFS repository, the installation

Table 1: Simultaneous installation and publication times (s)

n	Ideal	Current	Simultaneous
1	966	966	935
2	998	1964	977
3	1179	3143	1165
4	1179	4319	1183

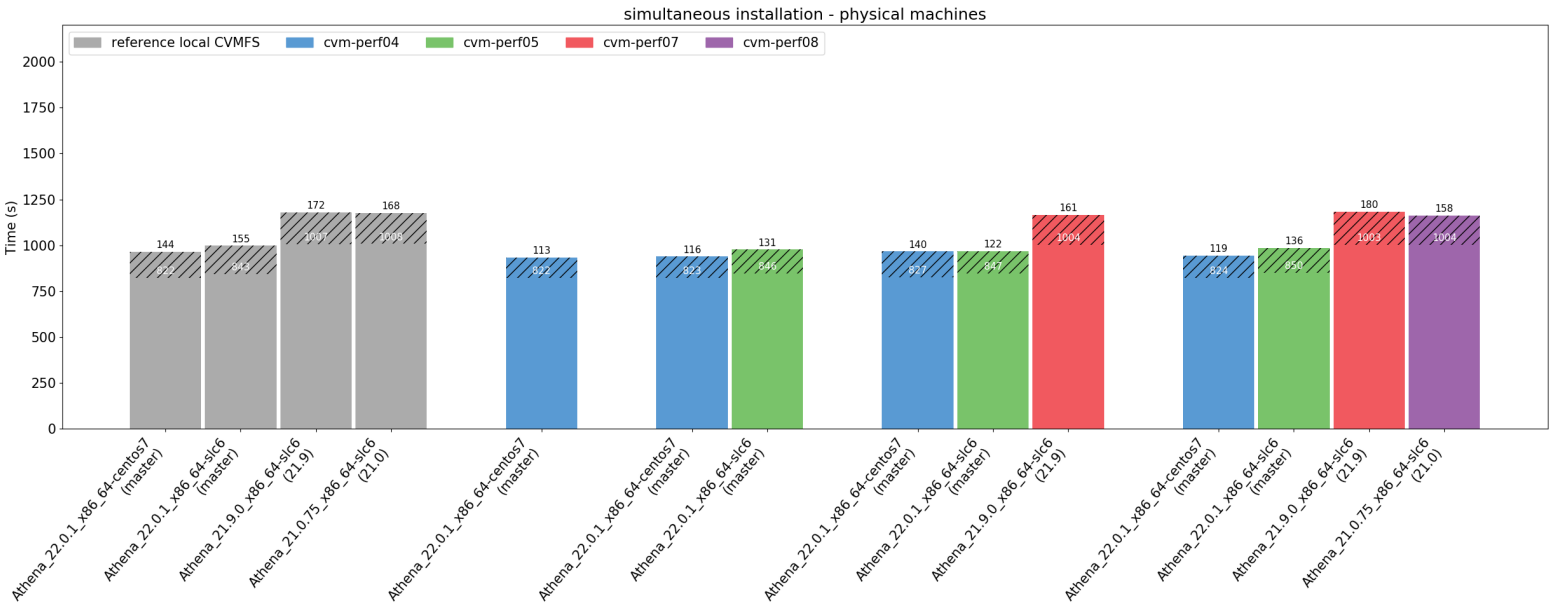


Figure 4: Simultaneous publications

and publication was measured separately. And not just the installation, but also the publication from all the RM machines was started at the same time – this simulated the worst case (and slowest) that could happen. After the simultaneous test, we did also test the functionality of the installed Athena release and everything worked as expected.

With these impressive results and the fact that this new functionality seems to be very stable now – no bad event happened during the tests – this new RM-GW approach is being considered for the production environment for publication of nightly Athena releases.

3 Manual pages for cvmfs_server

During my work I noticed there are no manual pages for the cvmfs_server utility. And because I had already written the Ansible playbook and the tests were running of their own, I had some time to work on the manual pages.

Previously (before my intervention) the only help for the cvmfs_server utility was a usage guide printed to the terminal standard output when no argument is provided or the argument syntax is incorrect. This usage output is quite long and required scrolling in the terminal or using a utility like less/more.

The relevant part of the code is written in shell script and the usage text is one long string. I decided to split it and place the documentation text before each subcommand function declaration. The example for the check subcommand is:

```
_CVMFS_DOC_CHECK_SHORT="Checks if the repository is sane"
_CVMFS_DOC_CHECK_SYNOPSIS="[options] <fqrn>"
_CVMFS_DOC_CHECK_DESCRIPTION="TODO"
_CVMFS_DOC_CHECK_OPTIONS="\
c:disable data chunk existence check
i:check data integrity (may take some time)
r:repair reflog problems
s:path to nested catalog subtree to check
t:tag (check given tag instead of trunk)"

cvmfs_server_check() {
  ...
}
```

The documentation variable names are composed of the prefix `_CVMFS_DOC_` followed by subcommand name and ending with one of `SHORT`, `SYNOPSIS`,

DESCRIPTION, OPTIONS, EXAMPLES, or SEE ALSO.

This allows one to have the relevant information close to the code, where it is expected and the manual pages are generated from those variables. There is one manual page for the cvmfs_server utility, but also all the subcommands have their own manual pages. With my changes it is possible to do e.g., `$ man cvmfs_server mkfs`.

But that is not all, with this approach it was easy to add help for each subcommand printed to STDOUT. There are two ways to call it. Either `$ cvmfs_server help <subcommand>`, or `$ cvmfs_server <subcommand> --help`. The user may choose depending on their personal preferences. An example is:

```
$ cvmfs_server help check
cvmfs_server check - Checks if the repository is sane

Synopsis: cvmfs_server check [options] <fqrn>

Options:
    -c : disable data chunk existence check
    -i : check data integrity (may take some time)
    -r : repair reflog problems
    -s : path to nested catalog subtree to check
    -t : tag (check given tag instead of trunk)
```