ARTEMIS

Test and Automation Infrastructure
Manual

Steffen Schwigon, Maik Hentsche

Table of Contents

1 Synopsis

ARTEMIS is an infrastructure.

It consists of applications, tools and protocols for testing software and evaluating the results. One focus is on testing Operating Systems in virtualization environments on AMD hardware.

There are 3 important layers:

- **Report Framework**
- **Test Suites**
- **Automation System**

The layers work completely autonomously, though can also be connected together.

To fully exploit the system the tasks you need to learn are

- **Connect and prepare a new machine into the infrastructure**
- **Write tests using the Test Anything Protocol (TAP)**
- **Write preconditions to describe automation tasks**
- **Review results via Web interface**
- **Evaluate results via Report Query interface**

Person in charge: Steffen Schwigon

2 Technical Infrastructure

2.1 Adding a new host into automation

This chapter describes what you need to do in order to get a new machine into the Artemis test rotation.

2.1.1 Make machine remote restartable

In the osrc network this means attaching it to `osrc_rst` which is the reset switch tool, a physical device plus the software to trigger the reset.

Person in charge: Jan Krocker

2.1.2 Make machine PXE boot aware

- Set booting order in BIOS to network first
- Create (or change) a file wotan:/tftpboot/cfgs/FOOBAR.lst
- Insert the following as first host configuration, i.e. after serial and timeout settings

  ```
  title Automatic test
  tftpserver 165.204.15.71
  configfile (nd)/tftpboot/FOOBAR.lst
  ```

  The IP address is that of our application server `bancroft`.
- Set the appropriate DHCP config in wotan:/etc/dhcpd.conf
- Search for your host in this file
- Add this line inside the config block of your host:

  ```
  option configfile "/tftpboot/cfgs/FOOBAR.lst";
  ```
- Force the dhcp server to reread its configuration with

  ```
  kill -HUP $(pidof dhcpd)
  ```

Person in charge: Maik Hentsche

2.1.3 Add host to the hardware database

If not already listed at `http://bancroft.amd.com/hardwaredb/` contact Jan Krocker.

Person in charge: Jan Krocker

2.1.4 Optionally: enable 'temare' to generate tests for this host

The steps until here are generally enough to put '`preconditions`' for this host into the Artemis database and thus use the host for tests.

Anyway, you can additionally register the host in '`temare`'.

temare is the *Test Matrix Replacement* program that schedules tests according to our test plan. If you want tests scheduled for the new machine then follow these steps:

- Login as root on bancroft
- Set the `PYTHONPATH` to include the *temare* src directory

  ```
  export PYTHONPATH=$PYTHONPATH:/home/artemis/temare/src
  ```
- Add the host to temare hostlist

  ```
  /home/artemis/temare/temare hostadd $hostname $memory $cores $bitness
  ```
- Add the host to `/home/artemis/temare/xentest.pl`

Person in charge: Maik Hentsche, Frank Arnold

3 Test Protocol

3.1 Test Anything Protocol (TAP)

3.2 Tutorial

3.2.1 Just plan and success

Example:

```
1..3
ok
ok
not ok
```

3.2.2 Succession numbers

Example:

```
1..3
ok 1
ok 2
not ok 3
```

- Missing lines can be detected.

3.2.3 Test descriptions

Example:

```
1..3
ok 1 - input file opened
ok 2 - file content
not ok 3 - last line
```

- Readability.

3.2.4 Mark tests as TODO

Example:

```
1..3
ok 1 - input file opened
ok 2 - file content
not ok 3 - last line # TODO
```

- mark not yet working tests as "TODO"
- allows test-first development
- "ok" TODOs can be recognized ("unexpectedly succeeded")

3.2.5 Comment TODO tests with reason

Example:

```
1..3
ok 1 - input file opened
ok 2 - file content
not ok 3 - last line # TODO just specced
```

- comment the TODO reason

3.2.6 Mark tests as SKIP (with reason)

Example:

```
1..3
ok 1 - input file opened
ok 2 - file content
ok 3 - last line # SKIP missing prerequisites
```

- mark tests when not really run (note the \u201cok\u201d)
- keeps succession numbers in sync

3.2.7 Diagnostics

Example:

```
1..3
ok 1 - input file opened
ok 2 - file content
not ok 3 - last line # TODO just specced
# Failed test 'last line'
# at t/data_dpath.t line 410.
# got: 'foo'
# expected: 'bar'
```

- Details

3.2.8 YAML Diagnostics

Example:

```
1..3
ok 1 - input file opened
ok 2 - file content
not ok 3 - last line # TODO just specced
  ---
  message: Failed test 'last line' at t/data_dpath.t line 410.
  severity: fail
  data:
    got: 'foo'
    expect: 'bar'
  ...
```

- allows parsable diagnostics
- we use that to track values inside TAP
- have a leading test line with number+description
- track complete data structures according to it
  - e.g., benchmark results

3.2.9 Headers for ARTEMIS

Example:

```
1..3
# Artemis-Suite-Name: Foo-Bar
# Artemis-Suite-Version: 2.010013
ok 1 - input file opened
ok 2 - file content
not ok 3 - last line # TODO just specced
```

```
@end example

@itemize
@item we use diagnostics lines (‘‘hot comments’’)
@item semantics only to our TAP applications
@end itemize

These are the headers that apply to the whole report:

@verbatim
 # Artemis-suite-name:                -- suite name
 # Artemis-suite-version:             -- suite version
 # Artemis-machine-name:              -- machine/host name
 # Artemis-machine-description:       -- more details to machine
 # Artemis-starttime-test-program:    -- start time for complete test
                                         (including guests)
 # Artemis-endtime-test-program:      -- end time for complete test
                                         (including guests)
 # Artemis-reportgroup-testrun:       -- associate this report with other
                                         reports of same testrun_id
 # Artemis-reportgroup-arbitrary:     -- associate this report with other
                                         reports of same arbitrary id
                                         (can be any string, but should be
                                         unique between all groups of the db,
                                         eg., an md5-hash of common
                                         characteristics of all test of one
                                         group)
```

3.2.10 Sections for ARTEMIS

Example:

```
1..2
# Artemis-section: arithmetics
ok 1 add
ok 2 multiply
1..1
# Artemis-section: string handling
ok 1 concat
1..3
# Artemis-section: benchmarks
ok 1
ok 2
ok 3
```

- we recognize “sections”, each with its own plan
- allows structuring of results,
- better readability later in web interface

These are the headers that apply to single sections:

```
 # Artemis-explicit-section-start:   -- explicitely start a section now
                                        instead of autorecognition
```

```
# Artemis-ram:                      -- memory
# Artemis-cpuinfo:                  -- what CPU
# Artemis-uname:                    -- kernel information
# Artemis-osname:                   -- OS information
# Artemis-bios:                     -- BIOS information
# Artemis-flags:                    -- flags, usually linux kernel
# Artemis-changeset:                -- exact changeset of the currently tested software or
# Artemis-description:              -- more description of the currently tested software or
                                       e.g., if changeset is not enough
# Artemis-uptime:                   -- uptime, maybe the test run time
# Artemis-language-description:     -- for Software tests,
                                       like "Perl 5.10", "Python 2.5"
# Artemis-xen-version:              -- Xen version
# Artemis-xen-changeset:            -- particular Xen changeset
# Artemis-xen-dom0-kernel:          -- the kernel version of the dom0
# Artemis-xen-base-os-description:  -- more verbose OS information
# Artemis-xen-guest-description:    -- description of a guest
# Artemis-xen-guest-test:           -- the started test program
# Artemis-xen-guest-start:          -- start time of test
# Artemis-xen-guest-flags:          -- flags used for starting the guest
# Artemis-kvm-module-version:       -- version of KVM kernel module
# Artemis-kvm-userspace-version:    -- version of KVM userland tools
# Artemis-kvm-kernel:               -- version of kernel
# Artemis-kvm-base-os-description:  -- more verbose OS information
# Artemis-kvm-guest-description:    -- description of a guest
# Artemis-kvm-guest-test:           -- the started test program
# Artemis-kvm-guest-start:          -- start time of test
# Artemis-kvm-guest-flags:          -- flags used for starting the guest
# Artemis-flags:                    -- Flags that were used to boot the OS
# Artemis-reportcomment:            -- Freestyle comment
```

3.2.11 Explicit section markers with lazy plans

In TAP it is allowed to print the plan (1..n) after the test lines (a "lazy plan"). In our ARTEMIS environment with concatenated sections this would break the default section splitting which uses the plan to recognize a section start.

If you want to use such a "lazy plan" in your report you can print an ARTEMIS header `Artemis-explicit-section-start` to explictely start a section. Everything until the next header `Artemis-explicit-section-start` is building one section. This also means that if you used this header **once** in a report you need to use it for **all** sections in this report.

The `Artemis-explicit-section-start` typically ignores its value but it is designed anyway to allow any garbage after the value that can help you visually structure your reports because explicit sections with "lazy plans" make a report hard to read.

Example:

```
# Artemis-explicit-section-start: 1 ------ arithmetics -------
# Artemis-section: arithmetics
ok 1 add
ok 2 multiply
1..2
# Artemis-explicit-section-start: 1 ------ string handling -------
# Artemis-section: string handling
```

```
ok 1 concat
1..1
# Artemis-explicit-section-start: 1 ------ benchmarks -------
# Artemis-section: benchmarks
ok 1
ok 2
ok 3
1..3
```

3.2.12 Developing with TAP

- TAP::Parser
  - `prove` tool
  - overall success and statistics
  - allows '`formatters`'
  - used to produce web reports

```
$ prove t/*.t
t/00-load.........ok
t/boilerplate.....ok
t/pod-coverage....ok
All tests successful.
Files=4, Tests=6, 0 wallclock secs
( 0.05 usr 0.00 sys + 0.28 cusr 0.05 csys = 0.38 CPU)
Result: PASS
```

3.2.13 TAP tips

- Easy to produce but using it **usefully** can be a challenge
- think "ARTEMIS" – l'Art 'emis – "The art to emit"
- use invariable test descriptions
- put meta information in diagnostics lines, not test descriptions
- use the description after `# TODO/SKIP`
- cheat visible (or: don't cheat invisible)
- really use `# TODO/SKIP`
- This keeps TAP evaluation consistent

3.3 Particular use-cases

3.3.1 Report Groups

3.3.1.1 Report grouping by same testrun

If we have a Xen environment then there are many guests each running some test suites but they don't know of each other.

The only thing that combines them is a common testrun-id. If each suite just reports this testrun-id as the group id, then the receiving side can combine all those autonomously reporting suites back together by that id.

So simply each suite should output

```
# Artemis-reportgroup-testrun: 1234
```

with 1234 being a testrun ID that is available via the environment variable `$ARTEMIS_TESTRUN`. This variable is provided by the automation layer.

3.3.1.2 Report grouping by arbitrary idenitifier

If the grouping id is not a testrun id, e.g., because you have set up a Xen environment without the ARTEMIS automation layer, then generate one random value once in dom0 by yourself and use that same value inside all guests with the following header:

- get the value:

  ```
  ARTEMIS_REPORT_GROUP=‘date|md5sum|awk ’{print $1}’‘
  ```

- use the value:

  ```
  # Artemis-reportgroup-arbitrary: $ARTEMIS_REPORT_GROUP
  ```

How that value gets from *dom0* into the guests is left as an exercise, e.g. via preparing the init scripts in the guest images before starting them. That’s not the problem of the test suite wrappers, they should only evaluate the environment variable `ARTEMIS_REPORT_GROUP`.

Person in charge: Frank Becker

4 Test Suite Wrappers

This section is about the test suites and wrappers around existing suites. These wrappers are part of our overall test infrastructure.

It's basically about the middle part in the following picture:

[[image:artemis_architecture_overview.png | 800px]]

We have wrappers for existing test and benchmark suites.

Wrappers just run the suites as a user would manually run them but additionally extract results and produce TAP (Test Anything Protocol).

We have some specialized, small test suites that complement the general suites, e.g. for extracting meta information or parsing logs for common problems.

If the environment variables

```
ARTEMIS_REPORT_SERVER
ARTEMIS_REPORT_PORT
```

are set the wrappers report their results by piping their TAP output there, else they print to STDOUT.

4.1 Available test suite wrappers

4.1.1 LMbench

artemis_testsuite_lmbench.sh
: A wrapper around the benchmark suite *LMbench*.

  See also http://www.bitmover.com/lmbench/.

4.1.2 kernbench

artemis_testsuite_kernbench.sh
: A wrapper around the benchmark suite *kernbench*.

  See also http://freshmeat.net/projects/kernbench/.

4.1.3 CTCS

artemis_testsuite_ctcs.sh
: A wrapper around the *Cerberus Test Control System (CTCS)*.

  See also http://sourceforge.net/projects/va-ctcs/.

4.1.4 LTP

artemis_testsuite_ltp.sh
: A wrapper around the *Linux Test Project (LTP)*.

  See also http://ltp.sourceforge.net/.

4.1.5 dom0-meta

artemis_testsuite_dom0_meta.sh
: A suite that produces meta information about the *dom0* environment.

4.2 Environment variables

The ARTEMIS automation layer provides some environment variables that the wrappers can use:

ARTEMIS_TESTRUN
: Currently active Testrun ID.

ARTEMIS_SERVER
: The controlling automation Server that initiated this testrun.

ARTEMIS_REPORT_SERVER
: The target server to which the tests should report their results in TAP.

ARTEMIS_REPORT_PORT
: The target port to which the tests should report their results in TAP. Complements `ARTEMIS_REPORT_SERVER`.

ARTEMIS_REPORT_API_PORT
: The port on which the more sophisticated Remote Reports API is available. It's running on the same host as `ARTEMIS_REPORT_SERVER`.

ARTEMIS_TS_RUNTIME
: Maximum runtime after which the testprogram will not be restarted when it runs in a loop. (This is a more passive variant than a timeout.)

ARTEMIS_GUEST_NUMBER
: Virtualisation guests are ordered, this is the guest number or 0 if not a guest.

ARTEMIS_NTP_SERVER
: The server where to request NTP dates from.

These variables should be used in the TAP of the suite as *Artemis*headers. Important use-case is "report groups", see next chapter.

Person in charge: Frank Becker

5 Preconditions

The central thing that is needed before a test is run is a so called *precondition*. Creating those preconditions is the main task needed to do when using the automation framework.

Most of the *preconditions* describe packages that need to be installed. Other preconditions describe how subdirs should be copied or scripts be executed.

A *precondition* can depend on other preconditions, leading to a tree of preconditions that will be installed from the leaves to the top.

5.1 SYNOPSIS

- Create a (maybe temporary) file
- Define conditions for a testrun: the *preconditions*
- Put the precondition into the database, maybe referring to other preconditions
- Create a testrun in the database, referring to the precondition
- Wait until the tesrun is executed and results are reported

5.2 Precondition repository

5.2.1 Normal preconditions

We store preconditions in the database and assign *testruns* to them (also in the database).

Usually the preconditions were developed in a (temporary) file and then entered into the database with a tool. After that the temporary file can be deleted. Note that such a precondition file can contain multiple precondition as long as they are formated as valid YAML.

Preconditions can be kept in files to re-use them when creating testruns but that's not needed for archiving purposes, only for creation purposes.

5.2.2 Macro preconditions

Though, there is another mechanism on top of normal preconditions: *Macro Preconditions*. These allow to bundle several preconditions into a common use-case and mark placeholders in them, see See [Macro Preconditions], page 19.

These *macro preconditions* should be archived, as they are only template files which are rendered into final preconditions. Only the final preconditions are stored in the database.

Macro preconditions can be stored in

`/data/bancroft/artemis/live/repository/macropreconditions/`

5.2.3 Precondition types

Some preconditions types can contain other more simple precondition types. To distinguish them we call them *Highlevel preconditions* and *Action preconditions*, accordingly.

5.2.3.1 Action preconditions

The following *action* precondition types are allowed:

package

A package (kernel, library, etc.), of type *.tar*, *.tar.gz* or *.tar.bz2*

image

A complete OS image of type *.iso*, *.tar.gz*, *.tgz*, *.tar*, *.tar.bz2*

prc

Create a config for the *PRC* module of the automation layer.

copyfile
: One file that can just be copied/rsync'd

installer_stop
: Don't reboot machine after system installer finished

grub
: Overwrite automatically generated grub config with one provided by the tester

fstab
: Append a line to /etc/fstab

repository
: Fetch data from a git, hg or svn repository

exec
: Execute a script during installation phase

reboot
: Requests a reboot test and states how often to reboot.

5.2.3.2 Highlevel preconditions

Currently only the following *high level* precondition type is allowed:

virt
: Generic description for Xen or KVM

High level preconditions both define stuff and can also contain other preconditions.

They are handled with some effort to *Do The Right Thing*, i.e., a defined root image in the high level precondition is always installed first. All other preconditions are installed in the order defined by its tree structure (depth-first).

5.2.4 Precondition description

We describe preconditions in YAML files (http://www.yaml.org/).

All preconditions have at least a key

```
precondition_type: TYPE
```

and optionally

```
name: VERBOSE DESCRIPTION
shortname: SHORT DESCRIPTION
```

then the remaining keys depend on the TYPE.

5.2.4.1 installer_stop

- stop run after system installer

  ```
  ---
  precondition_type: installer_stop
  ```

5.2.4.2 grub

- overwrite automatically generated grub config
  - Note: multiple lines in the grub file have to be given as one line separated by C<\n> in YAML
  - the variables $grubroot and $root are substituted with grub and /dev/* notation of the root partition respectively

- $root substitution uses the notation of the installer kernel. This may cause issues when the installer detects /dev/sd? and the kernel under test detects /dev/hd? or vice versa
- since grub always expects parentheses around the device, they are part of the substitution string for $grubroot
- note the syntax, to get multiline strings in YAML you need to start them with | and a newline

```
---
precondition_type: grub
config: |
  title Linux
    root $grubroot
    kernel /boot/vmlinuz root=$root"
```

5.2.4.3 package

- path names can be absolut or relative to /data/bancroft/artemis/development/repository/packages/
- supported packages types are rpm, deb, tar, tar.gz and tar.bz2
- package type is detected automatically
- absolute path: usually /data/bancroft/...
- relative path: relative to /data/bancroft/artemis/(live|development)/

```
---
filename: /data/bancroft/artemis/live/repository/packages/linux/linux-2.6.27.7.tar.bz2
precondition_type: package
```

5.2.4.4 copyfile

- a file that just needs to be scp or copied:
  - supported protocols are “scp”, “nfs” and “local”
  - the part before the first colon in the unique name is used as server name
  - the server name part is ignored for local
  - if dest ends in a slash, the file is copied with its basename preserved into the denoted directory
  - whether the “dest” is interpreted as a directory or a file is decided by the underlying “scp” or “cp” semantics, i.e., it depends on whether a directory already exists.

```
---
precondition_type: copyfile
protocol: nfs
source: osko:/export/image_files/official_testing/README
dest: /usr/local/share/artemis/perl510
```

5.2.4.5 fstab

- a line to add to /etc/fstab, e.g., to enable mounts once the system boots

```
---
precondition_type: fstab
line: "165.204.85.14:/vol/osrc_vol0 /home nfs auto,defaults 0 0"
```

5.2.4.6 image

usually the root image that is unpacked to a partition (this is in contrast to a guest file that’s just there)

- partition and mount are required, all other options are optional

- mount points are interpreted as seen inside the future installed system
- if no image is given, the already installed one is reused, i.e., only the mountpoint is mounted; make sure this is possible or your test will fail!
- can be either an iso file which is copied with dd or a tar, tar.gz or tar.bz2 package which is unpacked into the partition
- partitions are formated ext3 (only when image is given) and mounted to mount afterwards
  - this is why image exists at all, copyfile does not provide this
  - absolute "image": absolute
  - relative "image": relative to /data/bancroft/artemis/{live|development}/repository/images
    If not given, then it re-uses the partition without formatting/unpacking it.
  - partition: Can be /dev/xxx or LABEL or UUID.

    ```
    ---
    precondition_type: image
    mount: /
    partition: testing
    image: /data/bancroft/artemis/{live|development}/repository/images/rhel-5.2-rc2-32bi
    ```

5.2.4.7 repository

- git and hg are supported
- type and url are mandatory, target and revision are optional
- target denotes the directory where the source is placed in, the leading slash can be left out (i.e. paths can be given relative to root directory "'/"')

  ```
  ---
  precondition_type: repository
  type: git
  url: git://git.kernel.org/pub/scm/linux/kernel/git/avi/kvm.git
  target: kvm
  revision: c192a1e274b71daea4e6dd327d8a33e8539ed937
  ```

5.2.4.8 type: prc

Is typically contained implicitely with the abstract precondition *virt*. But can also be defined explicitely, e.g., for kernel tests.

Creates config for PRC. This config controls what is to be run and started when the machine boots.

- guest number

  If it is a guest, for host system use 0.
- test_program

  startet after boot by the PRC
- runtime

  The wanted time, how long it runs, in seconds, this value will be used to set an environment variable `ARTEMIS_TS_RUNTIME`, which is used by the test suite wrappers.
- timeout_testprogram

  Time that the testprogram is given to run, at most, after that it is killed (SIGINT, SIGKILL).
- guests

  Only used for virtualization tests. Contains an array, one entry per guest which defines how a guest is started. Can be a SVM file for Xen or an executable for KVM.

```
precondition_type: prc
config:
     runtime: 30
     test_program: /bin/uname_tap.sh
     timeout_after_testprogram: 90
     guests:
      - svm:  /xen/images/..../foo.svm
      - svm:  /xen/images/..../bar.svm
      - exec: /xen/images/..../start_a_kvm_guest.sh
```

5.2.4.9 type: exec

Defines which program to run at the installation phase.

```
precondition_type: exec
filename: /bin/some_script.sh
options:
  - -v
  - --foo
  - --bar="hot stuff"
```

The quotes in this example are actually wrong but left in so you learn the following lesson:

5.2.4.10 quote subtleties

Please note some subtlety about quotes.

- This is YAML. And YAML provides its own way of quoting.

  So this

  ```
  precondition_type: exec
  filename: /bin/some_script.sh
  options:
    - --foo
  ```

  and this

  ```
  precondition_type: exec
  filename: /bin/some_script.sh
  options:
    - "--foo"
  ```

  are actually the same (the value is always: `--foo`) because quotes at the beginning and end of a YAML line are used by YAML. When you use quotes at other places like in

  ```
  precondition_type: exec
  filename: /bin/some_script.sh
  options:
    - --bar="hot stuff"
  ```

  then they are not part of the YAML line but part of the value, so this time the value is: `--bar="hot stuff"`.

- Quotes are not shell quotes.

  So if you used quotes and they are not YAML quotes but part of the value then you should know that they are **not** evaluated by a shell when `some_script.sh` is called, because we use `system()` without a shell layer to start it.

  That's why in above example the quoted value `"hot stuff"` (with quotes!) is given as parameter `--bar` to the program. This usually **not** what you want.

- Summary: Yo nearly never need quotes.

  This is good enough:

```
precondition_type: exec
filename: /bin/some_script.sh
options:
  - -v
  - --foo
  - --bar=hot stuff
```

5.2.4.11 type: reboot

Requests a reboot test and states how often to reboot.

Note: Reboot count of 1 actually means boot two times since the first boot is always counted as number 0.

```
precondition_type: reboot
count: 2
```

5.2.4.12 type: autoinstall

Install a system using autoinstall scripts. The filename denotes the grub config to be used. It is mandatory and can be given as absolut path or relative to /data/bancroft/.../repository/install_grub/. The optional timeout is measured in second. If its absent a default value is used.

```
precondition_type: autoinstall
filename: suse/SLES10SP3_x86_64.lst
timeout: 1800
```

5.2.4.13 type: testprogram

Define which test program to run. This way of defining a test program should be prefered to using the PRC type precondition. Only the **testprogram** precondition guarantees parsing that sets all internal Artemis variables correctly.

```
precondition_type: testprogram
runtime: 30
program: /bin/uname_tap.sh
timeout: 90
parameters:
- --verbose
```

5.2.4.14 type: virt

A virtualization environment.

- guest root always needs to name the file to mount since its not easy or even impossible to get this name for some ways to install the root image (like tar.gz packages or subdir)
- guest root and guest config are installed inside the host, guest preconditions are installed inside the guest image
- guests can be started with xm create $xenconf, evaluation of $kvmconf or executing the $execconf script, thus only one of these three must be provided

- "'Note"': virt instead of virtualisation is used to reduce confusion for users whether British English (virtualisation) or American English (virtualization) is expected
- key "arch" arch: linux64 | linux32 (needed for for artemis toolchain)

```
name: automatically generated Xen test
precondition_type: virt
host:
  preconditions:
```

```
    - filename: /data/bancroft/artemis/live/repository/packages/xen/builds/x86_64/xen-3.3-
      precondition_type: package
    - filename: /data/bancroft/artemis/live/repository/packages/artemisutils/sles10/xen_in
      precondition_type: package
    - filename: /bin/xen_installer_suse.pl
      precondition_type: exec
    root:
      precondition_type: image
      partition: testing
      image: /data/bancroft/artemis/live/repository/images/suse/suse_sles10_64b_smp_raw.ta
      mount: /
      arch: linux64
    testprogram:
      execname: /opt/artemis/bin/artemis_testsuite_dom0_meta.sh
      timeout_testprogram: 10800
  guests:
  - config:
      precondition_type: copyfile
      protocol: nfs
      name: bancroft:/data/bancroft/artemis/live/repository/configs/xen/001-sandschaki-123
      dest: /xen/images/
      svm: /xen/images/001-sandschaki-1237993266.svm
    root:
      precondition_type: copyfile
      protocol: nfs
      arch: linux64
      name: osko:/export/image_files/official_testing/redhat_rhel4u7_64b_up_qcow.img
      dest: /xen/images/
      mountfile: /xen/images/001-sandschaki-1237993266.img
      mounttype: raw
    testprogram:
      execname: /opt/artemis/bin/py_ltp
      timeout_after_testprogram: 10800
```

5.2.4.15 General precondition keys “mountfile”

These 2 options are possible in each precondition. With that you can execute the precondition inside guest images:

```
mountfile: ...
mountpartition: ...
mounttype: @TODO{is this the same as mountfile, mountpartition?}
```

- 1. only mountfile: eg. rawimage, file loop-mounted - 2. only mountpartition: then mount that partition - 3. image file with partitions: mount the imagefile and from that only the given partition

Person in charge: Maik Hentsche

5.3 Macro Preconditions

This section describes macro precondition files as they are stored in `/data/bancroft/artemis/live/repository/macropreconditions/`.

A macro precondition denotes a file containing one or multiple preconditions and additional TemplateToolkit code.

In most cases preconditions for similar tests will only differ in one or very few keys. Thus precondition files could easily be reused by only changing these few keys. This is made easier with using macro preconditions. The macro precondition file should contain all preconditions to be reused. All variable keys should be substituted by appropriate TemplateToolkit variables. When creating the new testrun actual values for these TemplateToolkit variables have to provided.

Macro preconditions are **not** stored in the database. They are only a tool to ease the creation of preconditions. Only the **resulting** preconditions are stored in database.

To make parsing macro preconditions easier required and optional fields can be named after a comment field in the first lines of the file after the keys `artemis-mandatory-fields` and `artemis-optional-fields` respectively as in the following example:

```
# artemis-mandatory-fields: id
# artemis-optional-fields: kernel
```

None of these `# artemis-*` headers are required. But they help on frontend (like the Web GUI), therefor if they are missing then some functionality in frontends may also be missing.

The values for the placeholders can be filled via

```
artemis-testrun new [all usual options] \
    --macroprecond=FILENAME \
     -DPLACEHOLDER1=VALUE1 \
     -DPLACEHOLDER2=VALUE2 \
     -DPLACEHOLDER3=VALUE3
```

The FILENAME is a complete filename with absolute path.

There is no restriction on TemplateToolkit code for variable substitution. The following example could be used to generate a default value for the precondition key id.

```
[% id = BLOCK %][% IF id %][% id %][%ELSE%]2009-06-29-perfmon[% END %][% END %]
```

5.3.1 A real live example: kernel boot test

- Macroprecondition

```
# artemis-mandatory-fields: kernel_version
# artemis-optional-fields: kernelpkg
---
arch: linux64
image: suse/suse_sles10_64b_smp_raw.tar.gz
mount: /
partition: testing
precondition_type: image
---
precondition_type: copyfile
name: /data/bancroft/artemis/live/repository/testprograms/uname_tap/uname_tap.sh
dest: /bin/
protocol: local
---
precondition_type: copyfile
name: /data/bancroft/artemis/live/repository/packages/artemisutils/kernel/gen_initrd.sh
dest: /bin/
protocol: local
---
[% kernelpkg = BLOCK %][% IF kernelpkg %][% kernelpkg %][%ELSE%]kernel/linux-[% kernel_v
precondition_type: package
filename: [% kernelpkg %]
```

```
---
precondition_type: exec
filename: /bin/gen_initrd.sh
options:
  - [% kernel_version %]
---
precondition_type: prc
config:
      runtime: 30
      test_program: /bin/uname_tap.sh
      timeout_testprogram: 90
```

- The test script

  The test script `uname_tap.sh` to which the macro precondition refers is just a shell script that examines uname output:

```
#! /bin/sh
echo "1..2"
echo "# Artemis-Suite-Name: Kernel-Boot"
echo "# Artemis-Suite-Version: 1.00"
echo "# Artemis-Machine-Name: " `hostname`

if [ x`uname` != xLinux ] ; then echo -n "not " ; fi
echo "ok - We run on Linux"

if uname -a | grep -vq x86_64 ; then echo -n "not " ; fi
echo "ok - Looks like x86_64"
```

- Command line

  Once you wrote the macro precondition and the test script all you need is this command line:

```
artemis-testrun new \
  --hostname=dickstone \
  --macroprecond=/data/bancroft/artemis/live/repository/macropreconditions/kernel/kernel
   -Dkernelpkg=perfmon-682-x86_64.tar.gz \
   -Dkernel_version=2.6.28-rc3
```

  or with some more information (owner, topic):

```
artemis-testrun new \
  --owner=mhentsc3 \
  --topic=Kernel \
  --hostname=dickstone \
  --macroprecond=/data/bancroft/artemis/live/repository/macropreconditions/kernel/kernel
   -Dkernelpkg=perfmon-682-x86_64.tar.gz \
   -Dkernel_version=2.6.28-rc3
```

Person in charge: Steffen Schwigon

5.4 Producers

Sometimes, parameters for preconditions shall be defined when the testrun, this precondition is assigned to, is choosen for execution. This might apply for example when you want to test the newest build of a certain package. Also in combination with autorerun testruns dynamic assignment of preconditions is useful. These testruns are reinserted into the database automatically as soon as the scheduler chooses them for execution. In this case dynamic precondition

assignment allows these rerun tests to differ slightly. Preconditions with dynamically assigned parameters are called lazy precondition (similar to the lazy evaluation technique).

Dynamic precondition assignment is implemented using Precondition Producers. A producer is a modul that is called by the scheduler for handling of lazy preconditions. To use a lazy preconditio the user has to assign a precondition of type "producer" to the testrun. This precondition has to contain the basename of an existing producer module and may contain additional parameters. The producer will substitute the "producer" precondition with a normal precondition that has values assigned to all parameters.

5.4.1 Lazy precondition

Lets assume for example that you want to include the newest kernel package into your test. This can be achieved with the existing "Kernel" producer. Instead of a precondition of type "package" with a certain filename you should assign the following precondition to your testrun.

```
precontition_type: producer
producer: Kernel
```

This precondition will be substituted with a package precondition that has the latest Sysint kernel build set as filename.

5.4.2 Producer API

Producer are modules loaded into the scheduler. Thus they need to be written in Perl and reside inside the `Artemis::MCP::Scheduler::PreconditionProducer` namespace. A producer has to implement a method "produce". This function gets a job object as first parameter and a hash containing all additional options from the precondition as second parameter. It suggested that each producer inherits from `Artemis::MCP::Scheduler::PreconditionProducer`. Producers hall return a hash that has the produced preconditions as YAML text assigned to the hash key `precondition_yaml`. An optional key `topic` allows the producer to set the topic for the test. If the hash key `error` is set, the associated error string is reported and the testrun is cancled. In this case the other hash keys are not evaluated.

6 Command line interface

6.1 SYNOPSIS

- Get host usage/scheduling overview
- Create hosts
- Create queues
- Create hosts/queue bindings

6.2 Scheduling: hosts, queues, jobs

6.2.1 Create new queue, new host, bind both together

- Show existing queues with priorities

```
artemis@bancroft:~> artemis-testrun listqueue -v
        10 |              AdHoc |  1000
        11 |      kernel_reboot |   100
         4 | xen-3.3-testing-32 |   100
         5 | xen-3.3-testing-64 |   100
         7 | xen-3.4-testing-32 |   100
         6 | xen-3.4-testing-64 |   100
         9 |    xen-unstable-32 |   100
         8 |    xen-unstable-64 |   100
```

- Create new queue *oprofile*

```
artemis@bancroft:~> artemis-testrun newqueue --name=oprofile --priority=200
12
```

- Create new host *bullock* and bind it to queue *oprofile*

```
artemis@bancroft:~> artemis-testrun newhost --name=bullock --queue=oprofile
10
```

- Show existing hosts

  Note that the new host *bullock* is initially deactivated.

```
artemis@bancroft:~> artemis-testrun listhost -v
         8 |  amarok | deactivated |   free
         1 |  athene |      active | in use
         9 |   azael | deactivated |   free
        10 | bullock | deactivated |   free | oprofile
         4 |    cook | deactivated |   free
         6 | incubus | deactivated |   free
         2 |  kobold |      active | in use
         5 |  lemure |      active | in use
         3 |   satyr |      active | in use
         7 |    uruk | deactivated |   free
```

- Activate host *bullock*

  Note that this command is ID based (bullock has id 10) because you can rename hosts.

```
artemis@bancroft:~> artemis-testrun updatehost --id=10 --active
10 | bullock | active | free | oprofile
```

- Again, show existing hosts

  Host *bullock* is now activated.

```
artemis@bancroft:~> artemis-testrun listhost -v
         8 |  amarok | deactivated |   free
         1 |  athene |      active | in use
         9 |   azael | deactivated |   free
        10 | bullock |      active |   free | oprofile
         4 |    cook | deactivated |   free
         6 | incubus | deactivated |   free
         2 |  kobold |      active | in use
         5 |  lemure |      active | in use
         3 |   satyr |      active | in use
         7 |    uruk | deactivated |   free
```

Done.

6.2.2 Change queue priority

- List existing queues

```
artemis@bancroft:~> artemis-testrun listqueue -v
        10 |              AdHoc |  1000
        11 |      kernel_reboot |   100
        12 |           oprofile |   200 | bullock
         4 | xen-3.3-testing-32 |   100
         5 | xen-3.3-testing-64 |   100
         7 | xen-3.4-testing-32 |   100
         6 | xen-3.4-testing-64 |   100
         9 |    xen-unstable-32 |   100
         8 |    xen-unstable-64 |   100
```

- Update queue

```
artemis@bancroft:~> artemis-testrun updatequeue --name=oprofile --priority=1000
12
```

- Again, list existing queues

```
artemis@bancroft:~> artemis-testrun listqueue -v
        10 |              AdHoc |  1000
        11 |      kernel_reboot |   100
        12 |           oprofile |  1000 | bullock
         4 | xen-3.3-testing-32 |   100
         5 | xen-3.3-testing-64 |   100
         7 | xen-3.4-testing-32 |   100
         6 | xen-3.4-testing-64 |   100
         9 |    xen-unstable-32 |   100
         8 |    xen-unstable-64 |   100
```

  Done.

Person in charge: Maik Hentsche

7 Web User Interface

The Web User Interface is a frontend to the Reports database. It allows to overview reports that came in from several machines, in several test suites.

It can filter the results by dates, machines or test suite, gives colorful (RED/YELLOW/GREEN) overview about success/failure ratios, allows to zoom into details of single reports.

To evaluate reported test results in a more programmatic way, have a look into the *DPath Query Language* that is part of the [Reports::API], page 27.

7.1 Usage

The main URL is

`http://osrc.amd.com/artemis`

7.2 Understanding Artemis Details

7.2.1 Part 1 Overview

- Go to `https://osrc.amd.com/artemis/reports`
- Click "Last weeks test reports", aka. `https://osrc.amd.com/artemis/reports/date/7`
- Below day "Wed Oct 7, 2009" find the line

  ```
  20856  2009-10-07  Topic-xen-unstable  satyr  PASS testrun 9617
  ```

  To find this report you probably need to go more back into the past than just 7 days, or you use the direct link below.
- Note that there are other reports in this group that are greyed-out, i.e. all report ids of this testrun are:

  ```
  20856    Topic-xen-unstable
  20855    LMBench
  20854    CTCS
  20852    Host-Overview
  20851    Hardwaredb Overview
  ```
- Note that something FAILed in the CTCS run (20854).
- What we know until here:
  - It is a test for Xen-unstable (Topic-xen-unstable)
  - The running of the guests+suites itself worked well (20856 PASS)
  - There were 2 guest runs:

    ```
    LMBench      satyr:celegorm.osrc.amd.com     PASS
    CTCS         satyr:eriador                   FAIL
    ```
- Click on the ID link "20856" aka. `https://osrc.amd.com/artemis/reports/id/20856`

7.2.2 Part 2 Details

- Here you see the details of this report 20856.

  You see:
  - green PASSED results for the "MCP overview". This means the starting and finishing of the guests worked.
  - attachments of console logs.
  - some links to more information (raw TAP report, preconditions)
- Note below the group of all the other reports, again it's the group of those IDs:

```
20856    Topic-xen-unstable
20855    LMBench
20854    CTCS
20852    Host-Overview
20851    Hardwaredb Overview
```

- The most meta information is in “20852 Host-Overview”.
- Click on the ID link “20852” aka. https://osrc.amd.com/artemis/reports/id/20852
- Now you see the details of “20852 Host-Overview” with lots of meta information as “Context”.

  You see:

```
Metainfo
    cpuinfo:  1x Family: 15, Model: 67, Stepping: 2
    ram:      3950 MB
    uptime:   0 hrs

XEN-Metainfo
    xen_dom0_kernel:  2.6.18.8-xen x86_64
    xen_base_os_description:  SUSE Linux Enterprise Server 10 SP2 (x86_64)
    xen_changeset:    20273:10cfcbef68ee
    xen_version:      3.5-unstable

guest_1_redhat_rhel5u4_32bpae_qcow
    xen_guest_description:    001-lmbench
    xen_guest_flags:
    xen_guest_start:

guest_2_suse_sles10_sp3_gmc_32b_up_qcow
    xen_guest_description:    002-ctcs
    xen_guest_flags:
    xen_guest_start:
```

- If you are interested in what went wrong in the CTCS run, click on ID link “20854” aka. https://osrc.amd.com/artemis/reports/id/20854
- Here you see
  - one RED bar in CTCS-results
  - several RED bars in var_log_messages

  You can click on them to unfold the details.

7.2.3 Part 3 Testrun

- Imagine that the testrun completely failed and no usable reports arrived in, except that primary one from the MCP, then you can use the link at the end of the line

```
20856  2009-10-07  Topic-xen-unstable  satyr  PASS testrun 9617
                                                ------------
```

- Click on that link “testrun 9617” aka. https://osrc.amd.com/artemis/testruns/id/9617
- That contains the description what was **planned** in this testrun, regardless of whether it succeeded.

  (That’s the main difference between the two complementary concepts “Testrun” vs. “Reports”. The “Testrun” contains the specification, the “Reports” contain the results.)

  You see:

```
Name                    Automatically generated Xen test
Host
    Architecture        linux64
    Root image          /suse_sles10_sp2_64b_smp_raw.tar.gz
    Test                metainfo
Guest number 1
    Architecture        linux32
    Root image          /redhat_rhel5u4_32bpae_qcow.img
    Test                py_lmbench
Guest number 2
    Architecture        linux32
    Root image          /suse_sles10_sp3_gmc_32b_up_qcow.img
    Test                py_ctcs
```

- That's it, basically.

8 Reports::API

8.1 Overview

There runs yet another daemon, the so called `Artemis::Reports::API`, on the same host where already the `TAP Receiver` runs. This '`Reports API`' is meant for everything that needs more than just dropping TAP reports to a port, e.g., some interactive dialog or parameters.

This `Artemis::Reports::API` listens on Port `7358`. Its API is modeled after classic unix script look&feel with a first line containing a description how to interpret the rest of the lines.

The first line consists of a shebang (`#!`), a *api command* and *command parameters*. The rest of the file is the *payload* for the *api command*.

The syntax of the '`command params`' varies depending on the '`api command`' to make each command intuitively useable. Sometimes they are just positional parameters, sometimes they look like the start of a HERE document (i.e., they are prefixed with `<<` as you can see below).

Person in charge: Steffen Schwigon

8.2 Raw API Commands

In this section the raw API is described. That's the way you can use without any dependencies except for the minimum ability to talk to a port, e.g., via `netcat`.

See section [artemis-api], page 31 for how to use a dedicated command line utility that makes talking to the reports API easier, but is a dependency that might not be available in your personal test environment.

8.2.1 upload - attach a file to a report

This api command lets you upload files, aka. attachments, to reports. These files are available later through the web interface. Use this to attach log files, config files or console output.

8.2.1.1 Synopsis

```
#! upload REPORTID FILENAME [ CONTENTTYPE ]
payload
```

8.2.1.2 Parameters

- REPORTID

  The id of the report to which the file is assigned
- FILENAME

  The name of the file
- CONTENTTYPE

  Optional MIME type; defaults to `plain`; use `application/octet-stream` to make it downloadable later in browser.

8.2.1.3 Payload

The raw content of the file to upload.

8.2.1.4 Example usage

Just `echo` the first api-command line and then immediately `cat` the file content:

```
$ ( echo "#! upload 552 xyz.tmp" ; cat xyz.tmp ) | netcat -w1 bascha 7358
```

8.2.2 mason - Render templates with embedded query language

To query report results we provide sending templates to the API in which you can use a query language to get report details: This api-command is called like the template engine so that we can provide other template engines as well.

8.2.2.1 Synopsis

```
#! mason debug=0 <<ENDMARKER
payload
ENDMARKER
```

8.2.2.2 Parameters

- debug=1

  If '`debug`' is specified and value set to 1 then any error message that might occur is reported as result content. If debug is omitted or false and an error occurs then the result is just empty.

- <<ENDMARKER

  You can choose any word instead of ENDMARKER which should mark the end of input, like in HERE documents, usually some word that is not contained in the template payload.

8.2.2.3 Payload

A mason template.

Mason is a template language, see `http://masonhq.com`. Inside the template we provide a function `reportdata` to access report data via a query language. See section [Query language], page 31 for details about this.

8.2.2.4 Example usage

This is a raw Mason template:

```
% my $world = "Mason World";
Hello <% $world %>!
% my @res = reportdata '{ "suite.name" => "perfmon" } :: //tap/tests_planned';
Planned perfmon tests:
% foreach (@res) {
   <% $_ %>
% }
```

If you want to submit such a Mason template you can add the api-command line and the EOF marker like this:

```
$ EOFMARKER="MASONTEMPLATE".$$
$ payload_file="perfmon_tests_planned.mas"
$ ( echo "#! mason <<$EOFMARKER" ; cat $payload_file ; echo "$EOFMARKER" ) \
    | netcat -w1 bascha 7358
```

The output of this is the rendered template. You can extend the line to save the rendered result into a file:

```
$ ( echo "#! mason <<$EOFMARKER" ; cat $payload_file ; echo "$EOFMARKER" ) \
    | netcat -w1 bascha 7358 > result.txt
```

The answer for this looks like this:

```
Hello Mason World!
Planned perfmon tests:
   3
```

```
4
17
```

8.3 Query language DPath

The query language, which is the argument to the `reportdata` as used embedded in the '`mason`' examples above:

```
reportdata '{ "suite.name" => "perfmon" } :: //tap/tests_planned'
```

consists of 2 parts, divided by the '`::`'.

We call the first part in braces *reports filter* and the second part *data filter*.

8.3.1 Reports Filter (SQL::Abstract)

The *reports filter* selects which reports to look at. The expression inside the braces is actually a complete `SQL::Abstract` expression (http://search.cpan.org/~mstrout/SQL-Abstract/) working internally as a `select` in the context of the object relational mapper, which targets the table `Report` with an active JOIN to the table `Suite`.

All the matching reports are then taken to build a data structure for each one, consisting of the table data and the parsed TAP part which is turned into a data structure via `TAP::DOM` (http://search.cpan.org/~schwigon/TAP-DOM/).

The *data filter* works then on that data structure for each report.

8.3.1.1 SQL::Abstract expressions

The filter expressions are best described by example:

- Select a report by ID

  ```
  { 'id' => 1234 }
  ```

- Select a report by suite name

  ```
  { 'suite_name' => 'oprofile' }
  ```

- Select a report by machine name

  ```
  { 'machine_name' => 'bascha' }
  ```

- Select a report by date

  Here the value that you want to select is a structure by itself, consisting of the comparison operator and a time string:

  ```
  { 'created_at' => { '<', '2009-04-09 10:00' }  }
  ```

8.3.1.2 The data structure

8.3.2 Data Filter (Data::DPath)

The data structure that is created for each report can be evaluated using the *data filter* part of the query language, i.e., everything after the `::`. This part is passed through to `Data::DPath` (http://search.cpan.org/~schwigon/Data-DPath/).

8.3.2.1 Data::DPath expressions

8.3.3 Optimizations

Using the query language can be slow. The biggest slowdown occurs with the '`ANYWHERE`' element `//`, again with several of them, because they span up a big search tree.

Therefore, if you know the depth of your path, try to replace the `//` with some `*` because that only spans up on the current step not every possible step, like this:

```
{ ... } :: //section/stats-proc-interrupts-before//tap//data/TLB";
{ ... } :: /results/*/section/stats-proc-interrupts-before/tap/lines/*/_children/*/data/TL
```

8.4 Client Utility `artemis-api`

There is a command line utility `artemis-api` that helps with using the API without the need to talk the protocol and fiddle with `netcat` by yourself.

8.4.1 help

You can aquire a help page to each sub command:

```
$ /home/artemis/perl510/bin/artemis-api help upload
```

prints

```
artemis-api upload --reportid=s --file=s [ --contenttype=s ]
   --verbose          some more informational output
   --reportid         INT; the testrun id to change
   --file             STRING; the file to upload, use '-' for STDIN
   --contenttype      STRING; content-type, default 'plain',
                      use 'application/octed-stream' for binaries
```

8.4.2 upload

Use it from the Artemis path, like:

```
$ /home/artemis/perl510/bin/artemis-api upload \
  --file /var/log/messages \
  --reportid=301
```

You can also use the special filename '`-`' to read from STDIN, e.g., if you need to pipe the output of tools like `dmesg`:

```
$ dmesg | /home/artemis/perl510/bin/artemis-api upload \
  --file=- \
  --filename dmesg \
  --reportid=301
```

8.4.3 mason

TODO

9 Complete Use-Cases

In this chapter we describe how the single features are put together into whole use-cases.

9.1 Automatic Xen testing

This is a description on how to run Xen tests with *Artemis*using `SLES10` with one `RHEL5.2` guest (64 bit) as an example.

The following mainly applies to **manually** assigning Xen tests. The SysInt team uses *temare* to automatically create the here described steps.

9.1.1 Paths

- Host **bancroft**: /data/bancroft/artemis/live/
- Host **osko**: /export/image_files/official_testing/

9.1.2 Choose an image for Dom0 and images for each guest

We use suse/suse_sles10_64b_smp_raw.tar.gz as Dom0 and

```
osko:/export/image_files/official_testing/raw_img/redhat_rhel5u2_64b_smp_up_small_raw.img
```

as the only guest.

The SuSE image is of precondition type image. Thus its path is relative to `/mnt/images` which has `bancroft:/data/bancroft/artemis/live/repository/images/` mounted.

The root partition is named in the section '`root`' of the Xen precondition. Furthermore, you need to define the destination partition to be Dom0 root. We use `/dev/sda2` as an example. The partition could also be named using its UUID or partition label. Thus you need to add the following to the dom0 part of the Xen precondition:

```
root:
  precondition_type: image
  mount: /
  image: suse/suse_sles10_64b_smp_raw.tar.gz
  partition: /dev/sda2
```

The RedHat image is of type '`copyfile`'.

It is copied from `osko:/export/image_files/official_testing/raw_img/` which is mounted to `/mnt/nfs` before.

This mounting is done automatically because the protocol type nfs is given. The image file is copied to the destination named as dest in the '`copyfile`' precondition. We use `/xen/images/` as an example. To allow the System Installer to install preconditions into the guest image, the file to mount and the partition to mount need to be named. Note that even though in some cases, the mountfile can be determined automatically, in other cases this is not possible (e.g. when you get it from a tar.gz package). The resulting root secition for this guest is:

```
root:
  precondition_type: copyfile
  name: osko:/export/image_files/official_testing/raw_img/redhat_rhel5u2_64b_smp_up_small
  protocol: nfs
  dest: /xen/images/
```

```
mountfile: /xen/images/redhat_rhel5u2_64b_smp_up_small_raw.img
mountpartition: p1
```

9.1.3 PRC configuration

PRC (Program Run Control) is responsible for starting guests and test suites.

9.1.3.1 Guest Start Configuration

Making PRC able to start Xen guests is very simple. Every guest entry needs to have a section named "config". In this section, a precondition describing how the config file is installed and a filename have to be given. As for guest images the file name is needed because it can't be determined in some cases. We use 001.svm installed via copyfile to /xen/images/001.svm. The resulting config section is:

```
config:
  precondition_type: copyfile
  name: /usr/share/artemis/packages/mhentsc3/001.svm
  protocol: local
  dest: /xen/images/
  filename: /xen/images/001.svm
```

9.1.3.2 Testsuite Configuration

You need to define, where you want which test suite to run. This can be done in every guest and the Dom0. In this example, the Dom0 and the single guest will run different testsuites. this chapter only describes the Dom0 test program. See the summary at the end for details on the guest test program.

The section testprogram consists of a precondition definition describing how the test suite is installed. In our example we use a precondition type package with a relative path name. This path is relative to "'/data/bancroft/artemis/live/repository/packages/"'. Since "'bancroft:/data/bancroft/"' is mounted to "'/data/bancroft/"' in the install system, this directory can be accessed at "'bancroft:/data/bancroft/artemis/live/repository/packages/"'.

Beside the precondition you need to define an execname which is the full path name of the file to be executed (remember, it can't be determined). This file is called in the root directory ("'/"') in the test system thus in case you need to use relative paths inside your test suite they need to be relative to this. The program may take parameters which are named in the optional array "'parameters"' and taken as is. The parameter is "'timeout_after_testprogram"' which allows you to define that your test suite shall be killed (and an error shall be reported) after that many seconds. Even though this parameter is optional, leaving it out will result in Artemis waiting forever if your test doesn't send finish messages. The resulting testprogram section looks like this:

```
testprogram:
  precondition_type: package
  filename: artemis-testsuite-system.tar.gz
  path: mhentsc3/
  timeout_after_testprogram: ~
  execname: /opt/system/bin/artemis_testsuite_system.sh
  parameters:
    - --report
```

9.1.4 Preconditions

Usually your images will not have every software needed for your tests installed. In fact the example images now do but for the purpose of better explanation we assume that we need to install dhcp, python-xml and bridge-utils in Dom0. Furthermore we need a script to enable network and console. At last we install the Xen package and a Xen installer package. These two are still needed on our test images. Package preconditions may have a "'scripts"' array attached that name a number of programs to be executed after the package was installed. This is used in our example to call the Xen installer script after the Xen package and the Xen installer package were installed. See the summary at the end for the resulting precondition section. The guest image only needs a DHCP client. Since this precondition is appended to the precondition list of the appropriate guest entry, the System Installer will automatically know that the guest image has to be mounted and the precondition needs to be installed inside relative to this mount.

9.1.5 Resulting YAML config

After all these informations are gathered, put the following YAML text into a file. We use /tmp/xen.yml as an example.

```
precondition_type: xen
name: SLES 10 Xen with RHEL5.2 guest (64 bit)
dom0:
  root:
    precondition_type: image
    mount: /
    image: suse/suse_sles10_64b_smp_raw.tar.gz
    partition: /dev/sda2
  testprogram:
    precondition_type: package
    filename: artemis-testsuite-system.tar.gz
    path: mhentsc3/
    timeout_after_testprogram: 3600
    execname: /home/artemis/x86_64/bin/artemis_testsuite_ctcs.sh
    parameters:
      - --report
  preconditions:
    - precondition_type: package
      filename: dhcp-3.0.3-23.33.x86_64.rpm
      path: mhentsc3/sles10/
    - precondition_type: package
      filename: dhcp-client-3.0.3-23.33.x86_64.rpm
      path: mhentsc3/sles10/
    - precondition_type: package
      filename: python-xml-2.4.2-18.7.x86_64.rpm
      path: mhentsc3/sles10/
    - precondition_type: package
      filename: bridge-utils-1.0.6-14.3.1.x86_64.rpm
      path: mhentsc3/sles10/
# has to come BEFORE xen because config done in here is needed for xens initrd
    - precondition_type: package
      filename: network_enable_sles10.tar.gz
      path: mhentsc3/sles10/
      scripts:
```

```
        - /bin/network_enable_sles10.sh
    - precondition_type: package
      filename: xen-3.2_20080116_1546_16718_f4a57e0474af__64bit.tar.gz
      path: mhentsc3/
      scripts: ~
    - precondition_type: package
      filename: xen_installer_suse.tar.gz
      path: mhentsc3/sles10/
      scripts:
        - /bin/xen_installer_suse.pl
# only needed for debug purpose
    - precondition_type: package
      filename: console_enable.tar.gz
      path: mhentsc3/
      scripts:
        - /bin/console_enable.sh
guests:
  - root:
      precondition_type: copyfile
      name: osko:/export/image_files/official_testing/raw_img/redhat_rhel5u2_64b_smp_up_s
      protocol: nfs
      dest: /xen/images/
      mountfile: /xen/images/redhat_rhel5u2_64b_smp_up_small_raw.img
      mountpartition: p1
      #       mountpartition: /dev/sda3 # or label or uuid
    config:
      precondition_type: copyfile
      name: /usr/share/artemis/packages/mhentsc3/001.svm
      protocol: local
      dest: /xen/images/
      filename: /xen/images/001.svm
    testprogram:
      precondition_type: copyfile
      name: /usr/share/artemis/packages/mhentsc3/testscript.pl
      protocol: local
      dest: /bin/
      timeout_after_testprogram: 100
      execname: /bin/testscript.pl
    preconditions:
      - precondition_type: package
        filename: dhclient-4.0.0-6.fc9.x86_64.rpm
        path: mhentsc3/fedora9/
```

9.1.6 Grub

For Xen to run correctly, the defaults grub configuration is not sufficient. You need to add another precondition to your test. System Installer will replace $root with the /dev/ notation of the root partition and $grubroot with the grub notation (including parenthesis) of the root partition. Put the resulting precondition into a file. We use /tmp/grub.yml as an example. This file may read like this:

```
precondition_type: grub
config: |
 serial --unit=0 --speed=115200
 terminal serial
 timeout 3
 default 0
 title XEN-test
   root $grubroot
   kernel /boot/xen.gz com1=115200,8n1 console=com1
   module /boot/vmlinuz-2.6.18.8-xen root=$root showopts console=ttyS0,115200
   module /boot/initrd-2.6.18.8-xen
```

9.1.7 Order Testrun

To order your test run with the previously defined preconditions you need to stuff them into the database. Fortunatelly there are commandline tools to help you with this job. They can be found at "'/home/artemis/perl510/bin/"'. Production server for Artemis is bancroft.amd.com. Log in to this server (as root, since user login hasn't been thoroughly tested). Make sure that $ARTEMIS_LIVE is set to 1 and /home/artemis/perl510/bin/ is at the beginning of your $PATH (so the correct perl will always be found). For each precondition you want to put into the database you need to define a short name. Call "'/home/artemis/perl510/bin/artemis-testrun newprecondition"' with the appropriate options, e.g. in our example:

```
/home/artemis/perl510/bin/artemis-testrun newprecondition --shortname=grub --condition_fil
/home/artemis/perl510/bin/artemis-testrun newprecondition --shortname=xen --condition_file
```

C<artemis-testrun> will return a precondition ID in each case. You will need those soon so please keep them in mind. In the example the precondition id for grub is 4 and for Xen its 5.

You can now put your test run into the database using `/home/artemis/perl510/bin/artemis-testrun new`. This expects a hostname, a test program and all preconditions. The test program is never evaluated and only there for historical reasons. Put in anything you like. root is not yet know to the database as a valid user. Thus you need to add `--owner` with an appropriate user. The resulting call looks like this:

```
/home/artemis/perl510/bin/artemis-testrun new \\
   --hostname=bullock --precondition=4 --precondition=5 \\
   --test_program=whatever --owner=mhentsc3
```

C<artemis-testrun> new has more optional arguments, one of them being –earliest. This option defines when to start the test earliest. It defaults to "now". When the requested time has arrived, Artemis will setup the system you requested and execute your test run. Stay tuned. When everything went well, you'll see test output soon. For more information on what is going on with Artemis, see /var/log/artemis-debug.

Person in charge: Maik Hentsche

10 Artemis Development

This chapter is dedicated not to end users but to Artemis development.

10.1 Repositories

Artemis is developed using *git*. There is one central repository to participate on the development

```
ssh://gituser@wotan/srv/gitroot/Artemis
```

and one mirrored public one:

```
git://osrc.amd.com/artemis.git
```

10.2 Starting/Stopping Artemis server applications

This chapter assumes all services are deployed, as described in [Deployment], page 40.

10.2.1 Live environment

The live environment is based on the host `bancroft` for all the server applications, like mysql db, Reports::Receiver, Reports::API, Web User Interface, MCP.

10.2.1.1 Web User Interface

The application is configured inside the Apache config and therefore only needs Apache to be (re)started. `/home/artemis` must be mounted.

```
$ ssh root@bancroft
$ rcapache2 restart
```

10.2.1.2 Reports::Receiver

```
$ ssh root@bancroft
$ /etc/init.d/artemis_reports_receiver_daemon restart
```

10.2.1.3 Reports::API

```
$ ssh root@bancroft
$ /etc/init.d/artemis_reports_api_daemon restart
```

10.2.2 Development environment

The development environment is somewhat distributed.

On host `bascha` there are mysql db, Reports::Receiver, Reports::API, Web User Interface.

The MCP is usually running on host `siegfried`, with a test target machine `bullock`.

10.2.2.1 Preparing an MCP host

```
$ sudo apt-get install inetutils-inetd
$ sudo apt-get install atftpd
$ sudo chmod 777 /var/lib/tftpboot/
$ sudo ln -s /var/lib/tftpboot /tftpboot
$ # in /etc/group add group ‘‘artemis’’ with same ID as NFS group ‘‘artemis’’: 55435
$ # add local user to this ‘‘artemis’’ group
$ # needed to access /data/bancroft/artemis
```

10.2.2.2 Web User Interface

The application is running with its own webserver on `bascha`:

```
$ ssh ss5@bascha

# kill running process
$ kill ‘ps auxwww|grep artemis_reports_web_server | grep -v grep | awk ’{print $2}’ | sort

# restart
$ sudo /etc/init.d/artemis_reports_web
```

10.2.2.3 Reports::Receiver

```
$ ssh ss5@bascha
$ sudo /etc/init.d/artemis_reports_receiver_daemon restart
```

10.2.2.4 Reports::API

```
$ ssh ss5@bascha
$ sudo /etc/init.d/artemis_reports_api_daemon restart
```

10.2.3 Logfiles

The applications write logfiles on these places:

- MCP (automation master control program)

  `/var/log/artemis-debug`

- Reports::Receiver

  `/var/log/artemis_reports_receiver_daemon_stdout.log`
  `/var/log/artemis_reports_receiver_daemon_stderr.log`

- Reports::API

  `/var/log/artemis_reports_api_daemon_stdout.log`
  `/var/log/artemis_reports_api_daemon_stderr.log`

10.3 Deployment

This chapter is a collection of instructions how to build the Artemis toolchain.

The whole deployment process should be supported by a common build system, however that is not yet completed but done via several self-written build steps.

10.3.1 Create and upload Python packages

This is usually done by a developer on some working state that is worth to be installed in the development or live environment.

- Go to the development subdirectory of the package (lmbench wrapper in this example)

  ```
  cd Artemis/src/TestSuite-LmBench-Python
  ```

- Call make to generate the package used for live or development

  ```
  $ make devel
  ```

  or

  ```
  $ make live
  ```

10.3.2 Create and upload Perl packages

For `Artemis::MCP`, `Artemis::PRC`, `Artemis::Installer`, `Artemis::Schema` and `Artemis::Config` the deployment also works via `make devel/live`:

- ```
  $ make devel
  ```
  or
  ```
  $ make live
  ```

For the other Perl libraries of the Reports framework follow these steps:

- Go to the subdirectory of the package (Reports API in this example)
  ```
  $ cd Artemis/src/Artemis-Reports-API
  ```
- Call the Perl build steps to generate a distribution
  - If `Module::Install` driven:
    ```
    $ perl Makefile.PL
    $ make
    $ make test
    $ make dist
    ```
  - If `Module::Build` driven:
    ```
    $ perl Build.PL
    $ ./Build
    $ ./Build test
    $ ./Build dist
    ```

  Version numbers are not incremented automatically (as it can be done with the Python wrappers). The VERSION upgrade needs to be done manually before publicly uploading a new version.
- Upload the package
  ```
  $ ./scripts/dist_upload_wotan.sh
  ```

10.3.3 Generate complete Artemis toolchain in opt-artemis package

The previous chapters described how to build packages based on an already prepared build environment.

If you need to start from scratch the following section applies.

Following are the steps to create a `opt-artemis.tar.gz` package in a mounted and chrooted image. It compiles Perl and Python, installs them under `/opt/artemis` and installs the Artemis libraries. For the Perl part it also installs all CPAN dependencies from a local mirror.

The resulting `/opt/artemis` subdir can be used to continuously upgrade the Artemis libs as described in the preceding sections.

- Login as User '`ss5`'
  ```
  $ ssh ss5@bascha
  ```
- Copy base raw image to `/tmp`

  64bit:
  ```
  $ cp .../redhat_rhel4u7_64b_smp_qcow.img /tmp/
  ```
  32bit:
  ```
  $ cp .../redhat_rhel4u7_32b_smp_qcow.img /tmp/
  ```
- These images have multiple images, mount them
  ```
  $ sudo losetup /dev/loop1 /tmp/redhat_rhel4u7_64b_smp_raw.img
  $ sudo kpartx -a /dev/loop1
  $ sudo mount /dev/mapper/loop1p2 /mnt
  ```
- Other images might have only one partition, mount them with
  ```

```
$ sudo mount -o loop /tmp/one_partion_image_raw.img /mnt
```

- Mount directories into the chroot

  In this example we need `~ss5` as source for Perl and Python builders and a bind mounted `/dev` to get a random seed for ssh (used to fetch source).

```
$ sudo mkdir -p /mnt/home/ss5
$ sudo mount -o bind /2home/ss5 /mnt/home/ss5
$ sudo mount -o bind /dev/ /mnt/dev
$ sudo mkdir /mnt/home/artemis
$ sudo mount loge:/artemis /mnt/home/artemis
$ sudo mount -t proc proc /mnt/proc
```

- Chroot into the image

  64bit:

```
$ sudo chroot /mnt bash -l
```

  32bit:

```
$ linux32 sudo chroot /mnt bash -l
```

- Install git

  64bit:

```
$ rpm -ivh \
  ftp://ftp.tu-chemnitz.de/pub/linux/fedora-epel/4/x86_64/git-core-1.5.3.6-2.el4.x86_64
  ftp://ftp.tu-chemnitz.de/pub/linux/fedora-epel/4/x86_64/perl-Git-1.5.3.6-2.el4.x86_64
```

  32bit:

```
$ rpm -ivh \
  ftp://ftp.tu-chemnitz.de/pub/linux/fedora-epel/4/i386/git-core-1.5.3.6-2.el4.i386.rpm
  ftp://ftp.tu-chemnitz.de/pub/linux/fedora-epel/4/i386/perl-Git-1.5.3.6-2.el4.i386.rpm
```

- Enable public key authentication for git

```
$ cp -r /home/ss5/.ssh/ /root/
```

- Bootstrap complete Artemis toolchain

```
$ cd /home/ss5/artemis-perl
$ ./bootstrap_artemis_perl.sh
```

- `bootstrap_artemis_perl.sh` needs the user password for sudo, type it in
- This creates the directory `/opt/artemis/` but without any current Artemis code
- Copy Perl modules into `/opt/artemis`

```
$ rsync -r /home/artemis/perl510/lib/site_perl/5.10.0/Artemis/ \
           /opt/artemis/lib/perl5/site_perl/5.10.0/Artemis/
```

- Go to `/home/artemis/PYTHONREPO`, execute `build_python.sh` there

```
$ cd /home/artemis/PYTHONREPO ./wrapper_install_opt.sh
```

- Pack opt file together

```
$ cd /mnt/mnt/artemis
$ sudo tar -czf /tmp/opt-artemis64_rh4.7.tar.gz opt
```

- Copy package to `/data/bancroft`

```
$ sudo cp \
  /tmp/opt-artemis64_rh4.7.tar.gz \
  /data/bancroft/artemis/live/repository/packages/artemisutils/opt-artemis64_rh4.7.tar.
```

Person in charge: Maik Hentsche, Conny Seidel, Steffen Schwigon

10.3.4 Installation of the Web User Interface

The web application itself is available via the NFS mounted `/home/artemis`. On the application server in the Apache webserver you only need to configure a Location for the path `/artemis`.

```
bancroft$ cat /etc/apache2/conf.d/artemis_reports_web.conf

Alias / /home/artemis/perl510/bin/artemis_reports_web_fastcgi_live.pl/
<LocationMatch /artemis[.]*>
       Options ExecCGI
       Order allow,deny
       Allow from all
       AddHandler fcgid-script .pl
</LocationMatch>
```

Additionally there is a reverse proxy configured on `osrc.amd.com` that points to the application server:

```
osrc$ cat /etc/apache2/conf.d/artemis_reverse_proxy.conf

ProxyRequests Off
<Proxy *>
    Order deny,allow
    Allow from all
</Proxy>
ProxyPass        /artemis http://bancroft/artemis
ProxyPassReverse /artemis http://bancroft/artemis
ProxyPass        /hardwaredb http://bancroft/hardwaredb
ProxyPassReverse /hardwaredb http://bancroft/hardwaredb
```

Person in charge: Steffen Schwigon

10.4 Upgrading a database schema

The database schema is maintained as description for the Object Relational Mapper *DBIx::Class* using some versioning and upgrading features.

Those features are accessible via the command line tool `artemis-db-deploy`. The basic principle is:

(We show it here for `ReportsDB` schema. Same applies for `TestrunDB`.)

- Maintain schema in `src/Artemis-Schema/lib/Artemis/Schema/ReportsDB/Result/*.pm`
- Upgrade schema version in `src/Artemis-Schema/lib/Artemis/Schema/ReportsDB.pm`
- Upgrade package version in `src/Artemis-Schema/lib/Artemis/Schema.pm`
- Install schema package on a development machine
- Create a difference/upgrade file relative to an old version

  ```
  cd src/Artemis-Schema/
  artemis-db-deploy makeschemadiffs \
                    --db=ReportsDB \
                    --fromversion=2.010021 \
                    --upgradedir=./upgrades/
  ```

  In this example the version `2.010021` is the existing version. Do this for every version that you will later upgrade. Usually that's just the last one, but maybe you have several machines with different versions and want to upgrade them to this new version, then you need to call above line with all those old –fromversion's.

Of course you also can upgrade them in single steps from any old version via all intermediate versions to the latest. This is probably the best solution anyway.

- Add the diff/upgrade files to revision control

```
git add ./upgrades/
git commit -m'Schema: db upgrade files'
```

- Copy the diff/upgrade files to the target machine

  For some reason, the –upgradedir option does not work on the upcoming `upgrade` command but only the default subdir `var/tmp`, so you always need to copy the upgrade files, but only discriminate between development machine or the live machine.

```
rsync --progress -rc ./upgrades/ /var/tmp/
```

  or

```
rsync --progress -rc ./upgrades/ bancroft:/var/tmp/
```

- Upgrade the schema.

  Which environment and therefore which db connection is used depends on the environment. On development machine I have set

```
export ARTEMIS_DEVELOPMENT=1
```

  Then you just call

```
artemis-db-deploy upgrade --db=ReportsDB
```

10.5 Environment variables

There are some environment variables used in several contexts. Some of them are set from the automation layer to support the testsuites, some of them are used to discriminate between development and live context and some are just auxiliary variables to switch features.

Keep in mind that the variable needs to be visible where the actual component is running, which is sometimes not obvious in the client/server infrastructure.

- ARTEMIS_TESTRUN

  Set by the automation layer for the test suites which in turn should use it in there reports.

- ARTEMIS_SERVER

  Set by the automation layer for the test suites. Specifies the controlling host which initiated the testrun.

- ARTEMIS_REPORT_SERVER

  Set by the automation layer for the test suites. Specifies to which server the reports should be sent.

- ARTEMIS_REPORT_PORT

  Set by the automation layer for the test suites. Specifies to which port the reports should be sent.

- ARTEMIS_REPORT_API_PORT

  Set by the automation layer for the test suites. Specifies on which port the reports interface is listening, which is used, for instance, for uploading files.

- ARTEMIS_TS_RUNTIME

  Set by the automation layer for the test suites. Specifies the expected time that the testsuite should run. (Some suites, although only taking 1 hour, are re-run again and again for a given timespan.)

- ARTEMIS_NTP_SERVER

  Set by the automation layer for the test suites. Specifies the NTP server that the suite should use as reference for time shifting tests.

- ARTEMIS_GUEST_NUMBER

  Set by the automation layer for the test suites inside guests. Specifies which number the guest is, so the suite can report it and later this number helps sorting out results and context.

- ARTEMIS_OUTPUT_PATH

  Set and used by the automation layer. Specifies where the automation layer stores files like, e.g., console logs which are later uploaded. Can be used by the test suites; all files that they store there are automatically uploaded at the end of the testrun.

- ARTEMIS_DEVELOPMENT

  Used by Artemis::Config to switch the config space. By this the whole context of every module that is accessing the config in the same environment is switched to either "live" or "development". If not set to a true value the LIVE context is used by default.

- ARTEMIS_REPORTS_WEB_PORT

  Used by the web user interface Artemis::Reports::Web. Specifies the port on which it is running, usually only important for development mode. Else it is accessed via the usual Apache HTTP port 80.

- ARTEMIS_REPORTS_WEB_RELOAD

  Used by the web user interface Artemis::Reports::Web. Specifies whether the web application restarts if it recognizes changes in its source files.

- ARTEMIS_REPORTS_WEB_LIVE

  Used by the web user interface Artemis::Reports::Web. Specifies whether the config context either "live" or "development". It is therefore similar to `ARTEMIS_DEVELOPMENT` but the web user interface is disconnected from the automation layer, even in the used config, and therefore using its own mechanism.

10.6 Files

10.6.1 Special files

- /tmp/ARTEMIS_FORCE_NEW_TAPDOM

  Used by the database layer. If this file exists the cached values of TAP evaluation are thrown away and regenerated. This might be neccessary when the Artemis::TAP::Harness or used sub parts like TAP::DOM changed.

  Do not forget to remove this file, because it dramtically slows down all database TAP-DOM access!

- /tmp/ARTEMIS_CACHE_CLEAR

  Used by the Reports API. If this file exists the used caches are flushed.

  Do not forget to remove this file, because it dramtically slows down all DPath queries.

  It is suggested to just temporarily create it, trigger the API once so it flushes the caches and immediately remove the file.

- /tmp/FileCache

  This is where the Reports API caches DPath requests. Delete it if you want to flush the cache.

10.6.2 PID files

- /tmp/artemis-reports-receiver-daemon.pid

  Contains the Process ID of the reports receiver daemon.

- /tmp/artemis-reports-api-daemon.pid
  Contains the Process ID of the reports api daemon.
- /tmp/artemis_mcp_runloopdaemon.pid
  Contains the Process ID of the MCP runloop daemon.

10.6.3 Log files

- /tmp/artemis_reports_web.log
  Contains log of the Web user interface if it runs with own http daemon, usually in DEVELOPMENT mode.
- /var/log/apache2/error_log
  Contains log of the Web user interface if it runs under Apache2, usually in LIVE mode.
- /tmp/artemis_mcp_stderr.log
  Contains STDERR of the Master Control Program (MCP).
- /tmp/artemis_mcp_stdout.log
  Contains STDOUT of the Master Control Program (MCP).
- /var/log/artemis_reports_api_daemon_stderr.log
  Contains STDERR of the reports api daemon.
- /var/log/artemis_reports_api_daemon_stdout.log
  Contains STDOUT of the reports api daemon.
- /var/log/artemis_reports_receiver_daemon_stderr.log
  Contains STDERR of the reports receiver daemon.
- /var/log/artemis_reports_receiver_daemon_stdout.log
  Contains STDOUT of the reports receiver daemon.

10.7 Image preparation

There are several assumptions about available features inside the os images.

- netcat
  There must be a `netcat` or `nc` available.
- /etc/artemis
  The file `/etc/artemis` must be writeable.
  This is normally the case, but on some systems (non-writeable qcow images) this requires that the file is a link to a file in another writeable mounted image.
- /opt/artemis
  There directory `/opt/artemis` must be a writeable.
  This is normally the case, but on some systems (non-writeable qcow images) this requires that the file is a link to a file in another writeable mounted image.

10.8 temare - use a local temare

```
mkdir ~/temare
rsync -a ~/local/projects/Artemis/src/temare/ /2home/ss5/temare/
cd ~/temare
vi src/vi config.py
mkdir -p config/{xen,kvm}
cp /data/bancroft/artemis/live/configs/temare/test-schedule.db .
```

```
./temare subjectlist
./temare subjectstate kvm 64 enable
./temare subjectlist
```

10.9 Host Forensics

10.9.1 Investigate a host

- after it was installed but something went wrong so it could not continue or
- after it ran a test and it waits

10.9.2 Console

If ssh does not work, try to use the console.

10.9.3 View Artemis spec

The spec what to run on that host is in */etc/artemis*.
In the following example you can see those details:

- testrun id 12430
- execute /bin/artemis-testsuite-oprofile.sh
- report result to bancroft:7357
- MCP server is bancroft:37776

```
bullock:~ # cat /etc/artemis
---
hostname: bullock
mcp_server: bancroft
paths:
  autoinstall:
    grubfiles: /data/bancroft/artemis/live/repository/autoinstall/grubfiles/
  base_dir: /mnt/target/
  grubpath: /data/bancroft/artemis/live/configs/tftpboot
  guest_mount_dir: /mnt/guests/
  image_dir: /data/bancroft/artemis/live/repository/images/
  localdata_path: /tftpboot/
  nfskernel_path: /tftpboot/
  nfsroot: 165.204.15.71:/data/bancroft/artemis/live/nfsroot/installation_base/
  output_dir: /data/bancroft/artemis/live/output/
  package_dir: /data/bancroft/artemis/live/repository/packages/
  prc_nfs_mountdir: /data/bancroft/
  temare_path: /home/artemis/temare
  testprog_path: /data/bancroft/artemis/live/testprogram/
port: 37776
prc_nfs_server: bancroft
report_api_port: 7358
report_port: 7357
report_server: bancroft
test_run: 12430
testprogram_list:
  - program: /bin/artemis-testsuite-oprofile.sh
    timeout: 90
times:
```

```
boot_timeout: 1200
installer_timeout: 3600
poll_intervall: 10
reschedule_time: 3600
test_runtime_default: 7200
```

10.9.4 Restart the Artemis scripts on a waiting machine

The machine just ran a test but something in the scripts needs to be fixed.

You can edit the files in */opt/artemis/bin/* and */opt/artemis/lib/* and then restart the whole testrun with:

```
umount /data/bancroft/
/etc/init.d/artemis
```

10.10 Troubleshooting

10.10.1 Got a packet bigger than 'max_allowed_packet' bytes

- Context: mysql, usually via a Perl DBI driver.
- Error message:

```
Artemis::Reports::DPath::Mason::render_template::exec(anon_comp):
DBIx::Class::DynamicDefault::update():
DBI Exception:
DBD::mysql::st execute failed:
Got a packet bigger than 'max_allowed_packet' bytes
 [for Statement "UPDATE report SET tapdom = ?, updated_at = ? WHERE ( id = ? )"
  with ParamValues: 0='$VAR1 = [
  {
    'section' => {
       'artemis-meta-information' => {
          'tap' => bless( {
            'parse_errors' => [],
               'tests_run' => 1,
               'version' => 13,
               'exit' => 0,
               'start_time' => '1241791634.64412',
               'skip_all' => undef,
               'lines' => [
                  {
                    ...', 1='2009-05-08 14:07:14', 2='6033']
 at /home/artemis/perl510/lib/site_perl/5.10.0/
    Artemis/Schema/ReportsDB/Result/Report.pm line 132

Stack:
  [/home/artemis/perl510/lib/site_perl/5.10.0/Carp/Clan.pm:213]
  [/home/artemis/perl510/lib/site_perl/5.10.0/DBIx/Class/Exception.pm:58]
  [/home/artemis/perl510/lib/site_perl/5.10.0/DBIx/Class/Schema.pm:1020]
  [/home/artemis/perl510/lib/site_perl/5.10.0/DBIx/Class/Storage.pm:122]
  [/home/artemis/perl510/lib/site_perl/5.10.0/DBIx/Class/Storage/DBI.pm:863]
  [/home/artemis/perl510/lib/site_perl/5.10.0/DBIx/Class/Storage/DBI.pm:1113]
  [/home/artemis/perl510/lib/site_perl/5.10.0/DBIx/Class/Storage/DBI.pm:608]
  [/home/artemis/perl510/lib/site_perl/5.10.0/DBIx/Class/Storage/DBI.pm:1123]
```

```
[/home/artemis/perl510/lib/site_perl/5.10.0/DBIx/Class/Storage/DBI.pm:1206]
[/home/artemis/perl510/lib/site_perl/5.10.0/DBIx/Class/Row.pm:325]
[/home/artemis/perl510/lib/site_perl/5.10.0/DBIx/Class/Relationship/CascadeActions.pm:
[/home/artemis/perl510/lib/site_perl/5.10.0/DBIx/Class/DynamicDefault.pm:117]
[/home/artemis/perl510/lib/site_perl/5.10.0/Artemis/Schema/ReportsDB/Result/Report.pm:
[/home/artemis/perl510/lib/site_perl/5.10.0/Artemis/Reports/DPath.pm:75]
[/home/artemis/perl510/lib/site_perl/5.10.0/Artemis/Reports/DPath.pm:62]
[/home/artemis/perl510/lib/site_perl/5.10.0/Artemis/Reports/DPath.pm:26]
[/virtual/artemis_reports_dpath_mason:23]
```

- Fix:

  Increase the buffer size in the mysql config, e.g., on `bancroft` in `/etc/my.cnf`:

  ```
  [mysqld]
  ...
  max_allowed_packet = 128M
  ...
  ```