

AMD Operating System Research Center

**Tapper**

**Test and Automation  
Infrastructure**

**Manual**



**Tapper**  
**Test and Automation**  
**Infrastructure**  
**Manual**

AMD Operating System Research Center  
Steffen Schwigon, Maik Hentsche  
Copyright 2008-2012



# Contents

<b>1</b>	<b>Overview</b>	<b>10</b>
1.1	Infrastructure . . . . .	10
1.1.1	Automation . . . . .	10
1.1.2	Scheduling . . . . .	10
1.1.3	Web Application . . . . .	11
1.1.4	Result Evaluation . . . . .	11
1.1.5	Testplan Support . . . . .	11
1.1.6	Support for writing tests . . . . .	11
1.2	Many use-cases . . . . .	11
1.3	Technology . . . . .	12
1.4	Correlation to autotest.kernel.org . . . . .	12
<b>2</b>	<b>Synopsis</b>	<b>12</b>
2.1	Tapper infrastructure . . . . .	12
2.2	Vocabulary . . . . .	13
2.2.1	Master Control Program (MCP) . . . . .	13
2.2.2	Program Run Control (PRC) . . . . .	13
2.2.3	Reports Receiver . . . . .	13
2.2.4	Reports API . . . . .	13
2.2.5	Web User Interface . . . . .	13
2.2.6	Reports DB . . . . .	14
2.2.7	Testrun DB . . . . .	14
2.2.8	Testrun . . . . .	14
2.2.9	Preconditions . . . . .	14
2.2.10	Report . . . . .	14
2.2.11	Test Anything Protocol (TAP) . . . . .	14
2.2.12	TAP archives . . . . .	14
<b>3</b>	<b>Technical Infrastructure</b>	<b>14</b>
3.1	Adding a new host into automation . . . . .	14
3.1.1	Make machine remote hard resetable . . . . .	15
3.1.2	Machine specific configuration . . . . .	15
3.1.3	Make machine PXE boot aware . . . . .	17
3.1.4	Configure TFTP on central MCP machine . . . . .	17
3.1.5	Make the hosts known in the TestrunDB . . . . .	18
3.1.6	Optionally: enable <code>temare</code> to generate tests for this host . . . . .	18
3.1.7	Optionally: Web server for autoinstall . . . . .	19
<b>4</b>	<b>Test Protocol</b>	<b>19</b>
4.1	Test Anything Protocol (TAP) . . . . .	19
4.2	Tutorial . . . . .	19
4.2.1	Just plan and success . . . . .	19
4.2.2	Succession numbers . . . . .	20
4.2.3	Test descriptions . . . . .	20
4.2.4	Mark tests as TODO . . . . .	20
4.2.5	Comment TODO tests with reason . . . . .	21

4.2.6	Mark tests as SKIP (with reason)	21
4.2.7	Diagnostics	21
4.2.8	YAML Diagnostics	22
4.2.9	Meta information headers for reports	22
4.2.10	Report sections	23
4.2.11	Meta information headers for report sections	24
4.2.12	Meta information structure summary	25
4.2.13	Explicit section markers with lazy plans	26
4.2.14	Developing with TAP	27
4.2.15	TAP tips	27
4.3	Particular use-cases	28
4.3.1	Report Groups	28
4.4	TAP Archives	28
4.5	Reporting TAP to the reports receiver	28
4.5.1	Submitting raw TAP	29
4.5.2	Submitting TAP Archives	29
<b>5</b>	<b>Test Suite Wrappers</b>	<b>29</b>
5.1	Available test suite wrappers	29
5.1.1	tapper-testsuite-autotest	30
5.2	Environment variables	30
<b>6</b>	<b>Preconditions</b>	<b>30</b>
6.1	SYNOPSIS	31
6.2	Layers of preconditions	31
6.3	Precondition repository	31
6.4	Overview Precondition Types	31
6.5	Details Precondition Types	31
6.6	Macro Preconditions	31
6.7	Testplans	31
6.8	Precondition Producers	31
6.9	Layers of preconditions	31
6.9.1	Normal preconditions	31
6.9.2	Macro preconditions	31
6.9.3	Testplans	32
6.10	Precondition repository	32
6.11	Overview Precondition Types	32
6.11.1	Overview: Action precondition types	32
6.11.2	Overview: Highlevel precondition types	33
6.12	Details Precondition Types	33
6.12.1	installer_stop	33
6.12.2	grub	33
6.12.3	package	34
6.12.4	copyfile	34
6.12.5	fstab	35
6.12.6	image	35
6.12.7	repository	35
6.12.8	type: prc	36

---

6.12.9	type: exec	36
6.12.10	type: hint	37
6.12.11	quote subtleties	37
6.12.12	type: reboot	38
6.12.13	type: autoinstall	38
6.12.14	type: testprogram	38
6.12.15	type: autoinstall	39
6.12.16	type: cobbler	39
6.12.17	type: virt	39
6.12.18	General precondition keys “mountfile”	40
6.13	Macro Preconditions	41
6.13.1	A real live example - kernel boot test	41
6.14	Testplans	43
6.15	Precondition Producers	45
6.15.1	Lazy precondition	45
6.15.2	Producer API	45
6.15.3	Existing producers	46
6.15.4	Example: “Kernel” precondition producer	46
<b>7</b>	<b>Command line interface</b>	<b>47</b>
7.1	Commandline Synopsis	47
7.2	Scheduling	47
7.2.1	Create new queue and new host and bind both together	47
7.2.2	Change queue priority	48
7.2.3	requested features	49
7.2.4	Cancel current testrun on host	50
<b>8</b>	<b>Web User Interface</b>	<b>50</b>
8.1	Usage	50
8.2	Understanding Tapper Details	50
8.2.1	Part 1 Overview	50
8.2.2	Part 2 Details	51
8.2.3	Part 3 Testrun	52
<b>9</b>	<b>Reports API</b>	<b>53</b>
9.1	Overview	53
9.2	Raw API Commands	53
9.2.1	upload aka. attach a file to a report	53
9.2.2	download - download a file which is attached to a report	54
9.2.3	mason - Render templates with embedded query language	54
9.3	Query language DPath	55
9.3.1	Reports Filter (SQL-Abstract)	56
9.3.2	Data Filter	56
9.3.3	Optimizations	56
9.4	Client Utility tapper-api	56
9.4.1	help	57
9.4.2	upload	57
9.4.3	mason	57

<b>10 Complete Use Cases</b>	<b>57</b>
10.1 Automatic Xen testing . . . . .	57
10.1.1 Paths . . . . .	57
10.1.2 Choose an image for Dom0 and images for each guest . . . . .	58
10.1.3 PRC configuration . . . . .	58
10.1.4 Preconditions . . . . .	59
10.1.5 Resulting YAML config . . . . .	60
10.1.6 Grub . . . . .	61
10.1.7 Order Testrun . . . . .	62



## 1 Overview

Tapper is an infrastructure for all aspects of testing, scaling from usage as simple test result database to managing complete test labs with testplan scheduling, machine install automation, and complex result evaluation.

### 1.1 Infrastructure

#### 1.1.1 Automation

- Network boot (PXE + TFTP + NFS)
- Machine setup from scratch driven by YAML specfiles
- Machine setup can be done by installing images, running kickstart/autoyast/preseed installer, or via Cobbler
- Machine setup supports virtualization (Xen and KVM)
- Alternatively using pre-installed machines via ssh
- Image-based or kickstart/autoyast based installation
- Lightweight status API to easily plug in foreign installers
- Inject packages, programs, files into hosts/guests
- Control execution of test scripts in hosts/guests
- Interdependent setup and sync of co-operating machines
- Complex timeout handling for complete test cycle, inclusive virtualized guest
- Reboot handling
- Console logging
- Hardware reset

#### 1.1.2 Scheduling

- Optimize utilization of pools with "not enough" machines for "too many use-cases"
- Multiplex queues of freely definable use-cases (e.g., Xen, KVM, baremetal kernels, ad-hoc tests, benchmarks) with different bandwidths
- Pluggable core scheduling algorithm (default "Weighted Fair Queuing")
- Feature driven host matching (by memory, cores, vendor, etc. or complex combinations)
- Dynamic queue and host management (add, delete, de/activate)
- Host/queue binding for dedicated scheduling
- Auto re-queuing for continuous testing
- Multi-host scenarios

### 1.1.3 Web Application

- Frontend to 2 databases: testruns and reports
- Providing a "management view" and high-level test result evaluation
- Overview lists and detailed zoom-in, GREEN/YELLOW/RED coloring
- Filters over time, testsuites, machines, success status
- RSS feeds on such filters
- Visualize groups of connected results (virtualized neighbour guests)
- Control start of testruns

### 1.1.4 Result Evaluation

- Programmable complement to the web frontend
- Allow complex queries on the test result database
- No client side toolchain necessary
- Easy API to send and get back queries embedded in templates
- API allows SQL and XPath like queries in an abstract way

### 1.1.5 Testplan Support

- Testplans combine many Tapper features into concise points of interest
- Manage testplan hierarchy allowing specfile reuse for complex testplan matrix
- Interoperation with TaskJuggler on dedicated QA tasks for automatic scheduling and reporting
- Consequently a complete focused QA life cycle of planning, execution and reporting

### 1.1.6 Support for writing tests

- Format test results based on standard Test Anything Protocol (TAP)
- Existing standard TAP toolchains available for about 20 programming languages
- Dedicated Tapper support for Shell, Perl, Python available

## 1.2 Many use-cases

- Kernel testing (stress, function, reboot)
- Virtualization Testing (Xen, KVM)
- Test matrix of host/guest combinations
- Distribution testing (like OS or compilers)
- Multi-machine scenarios
- Complex result evaluation

### 1.3 Technology

- Platform independent (primarily x86, but already seen on ARM)
- Test Anything Protocol (TAP)
- Core system written in Perl and CPAN
- DB independent via OR mapper, developed on MySQL and SQLite
- HTML/CSS, cautious Javascript
- Language agnostic testing (e.g, Perl/Python/Shell test suites)
- PXE, GRUB, TFTP, NFS boot automation
- optional Cobbler integration
- Strong decoupling of functional layers (webgui, testsuites, automation) to allow you to amalgamate your own infrastructure

### 1.4 Correlation to autotest.kernel.org

The main focus of `autotest.kernel.org` project is on testing the Linux kernel. It provides a broad coverage of kernel functionality testing and wrappers of many existing test suites.

**Tapper** provides many complex scenarios, like virtualization (Xen/KVM), distribution testing (RHEL, SLES, Debian, Ubuntu), SimNow testing and benchmarking. **Tapper** can schedule them all multiplexed according to "use-case bandwidths" over large or small machine pools.

The `autotest.kernel.org` client can be used in a **Tapper** infrastructure via a thin wrapper that utilizes the TAP export we provided to the `autotest` project.

**Tapper** then complements it with Testplan support, a result database and a homogeneous result evaluation API.

## 2 Synopsis

### 2.1 Tapper infrastructure

Tapper is an infrastructure.

It consists of applications, tools and protocols for testing software and evaluating the results. One initial main focus was on testing Operating Systems in virtualization environments. It is now a modular infrastructure for lots of other, related scenarios, like benchmarking or build systems.

There are 3 important layers:

- **Reports Framework**
- **Test Suites**
- **Automation System**

The layers can work completely autonomously, though can also be connected together and are targeted to be stacked in this order:

- The **Report Framework** is for receiving and evaluating test reports sent by any source. The only requirement is that the result reports are using TAP, the Test Anything Protocol.
- The **Test Suites** are either directed test suites or wrappers around existing test projects doing conversion of their results into TAP. These are the TAP producers that create reports and send them to the Reports Framework.
- The **Automation System** is taking care of setting up machines, installing dependencies and test suites and running the latter. It can set up virtualized environments.

To fully exploit the system you need to learn:

- **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**

## 2.2 Vocabulary

### 2.2.1 Master Control Program (MCP)

There is a central server controlling the automation by running the **Master Control Program**, aka. **MCP**. Usually it also centralizes several other services: it is the TFTP server for network booting, runs the daemons of the reports framework (reports receiver, remote api) and the web application, including the mysql databases, and also serves the file repository via NFS.

### 2.2.2 Program Run Control (PRC)

When machines run automated tests, these test program runs are controlled by a **Program Run Control**, aka. **PRC**. In virtualization scenarios, each host and guest has its own PRC, numbered PRC0 (for the host), PRC1 (1st guest), PRC2 (2nd guest), etc.

### 2.2.3 Reports Receiver

The **Reports Receiver** means the daemons that accept reports. We often run them on the same machine as the MCP and the Web framework, but that's not necessary.

### 2.2.4 Reports API

Similar to the reports receiver is the **Reports API** which is the daemon for all more complex interfaces, like uploading files, downloading files, querying the reports. Similar to reports API we often run them on the same machine as the MCP and the Web application, but that's not necessary.

### 2.2.5 Web User Interface

The **Web User Interface** is an independent web application. Similar to the reports receiver and the reports API it can run anywhere, either standalone or in Apache, via `mod_perl`, `FCGI`, etc.. The only common thing for all those central applications (MCP, reports receiver, reports api, web application) is the config to use the same databases.

#### 2.2.6 Reports DB

The **Reports DB** contains all data that are reported. It's the base for the reports receiver, the reports API, the web application.

#### 2.2.7 Testrun DB

The **Testrun DB** is the DB for the automation layer. It contains hosts, testrun specifications and scheduling information.

#### 2.2.8 Testrun

A **Testrun** is a request to the automation layer to set up a host machine and run a workload on it. It consists of “preconditions” and scheduling information (host name, wanted host features, scheduling queue).

#### 2.2.9 Preconditions

**Preconditions** are specifications that describe how to set up a host. They are the essential part of a testrun.

#### 2.2.10 Report

A **Report** is the reported result of any workload, regardless of how it was produced (automatically, by a tes suite, manually via echo and netcat). Its only requirement is to be formatted in TAP (the Test Anything Protocol), or as TAP archive.

#### 2.2.11 Test Anything Protocol (TAP)

The **Test Anything Protocol** aka. **TAP** is the syntax to describe test results.

#### 2.2.12 TAP archives

A **TAP archive** is a .tar.gz file that contains files of TAP. It's the result of a test suite that can consist of many parts compressed into a single file.

## 3 Technical Infrastructure

See also the “Getting Started Guide” for more complete step-by-step instructions how to install the infrastructure from scratch up to a first example test run.

### 3.1 Adding a new host into automation

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

### 3.1.1 Make machine remote hard resettable

Connect the machine physically to some facility to programmatically switch it completely off. This can be the Reset cable wires connected to a dedicated reset box which can be programmed usually with an external tool. It can also be a TCP/IP controllable Power Control. As an example Tapper comes with a plugin for the “Infratec PM211 MIP” ethernet controllable multi socket outlet. To use it write this in the configuration file:

```
reset_plugin: PM211MIP
reset_plugin_options:
  ip: 192.168.1.39
  user: admin
  passwd: secret
outletnr:
  johnconnor: 1
  sarahconnor: 2
```

This configures to use the PM211MIP plugin for reset and gives it the configuration that the host “johnconnor” is connected on port 0 and the host “sarahconnor” on port 1, together with IP address, username and password of the multi-socket outlet.

If you have other hardware then write your own reset plugin FooBar in a Perl module `Tapper::MCP::Net::Reset::FooBar`. Look into the code of `Tapper::MCP::Net::Reset::PM211MIP` to get inspiration.

### 3.1.2 Machine specific configuration

Tapper configuration happens in `/etc/tapper.cfg` which contains everything that needs to be overwritten from the default values that come with the library `Tapper::Config`.

Config entries for `grub` can be configured for defaults like this:

```
mcp:
  installer:
    default_grub: |
      serial --unit=0 --speed=115200
      terminal serial
      default 0
      timeout 2
      title Test run (Install)
      tftpserver $TAPPER_TFTPSERVER
      kernel $TAPPER_KERNEL root=/dev/nfs reboot=force ro ip=dhcp \
        nfsroot=$TAPPER_NFSROOT $TAPPER_OPTIONS $HOSTOPTIONS
test:
  default_grub: |
    default 0
    timeout 2
    title Test run (Execute)
    root $grubroot
    kernel /boot/vmlinuz root=$root reboot=force ip=dhcp noapic \
      $TAPPER_OPTIONS $HOSTOPTIONS
    $initrd_options
```

You can see two entries, one for the installer to boot from TFTP, and one for the actual test run after the machine was installed. Usually the second part (for test run) is defined in a precondition because it needs special needs.

In these grub entries the MCP substitutes several variables before the actual grub entry is written:

- `$grubroot`

Substituted with something like `(hd0,1)`.

- `$root`

Substituted with something like `/dev/sda2`.

- `$TAPPER_TFTPSERVER`

Substituted with tftpserver IP address as configured in `/etc/tapper.cfg`, like:

```
tftp_server_address:    165.204.15.222
```

- `$TAPPER_KERNEL`

Substituted with kernel image file as configured in `/etc/tapper.cfg`, like:

```
files:
# kernel for installer
installer_kernel: /tftpboot/bzImage
```

- `$TAPPER_NFSROOT`

Substituted with NFS root filesystem as configured in `/etc/tapper.cfg`, like:

```
paths:
# Path to installer nfs root as set in PXE grub config
# (IP of central host "tapper")
nfsroot: 165.204.15.222:/data/tapper/live/nfsroot/
```

- `$TAPPER_OPTIONS`

Substituted with something like `tapper_ip=165.204.15.222 tapper_port=1337 testrun=777777 tapper_host=tapper tapper_environment=live` (as one single line), where “tapper\_ip” and “tapper\_port” are the host and port to send status messages to, the “testrun” is the testrun id of the current run, “tapper\_host” a hostname alternative to “tapper\_ip” and “tapper\_environment” sets the context (“live” or “development”).

- `$HOSTOPTIONS`

This substitution is controlled by another entry in `/etc/tapper.cfg`:

```
grub_completion_HOSTOPTIONS:
_default:
earlyprintk=serial,ttyS0,115200 console=ttyS0,115200
capalus:
earlyprintk=serial,ttyS4,115200 console=ttyS4,115200
garcia:
OxAFFE
```

There you see a default value and two entries for the hosts “capalus” (defining a different serial interface for console) and “garcia” (defining something completely different without any console usage).

### 3.1.3 Make machine PXE boot aware

- Set booting order in BIOS to network first
- Configure DHCP for each connected machine

The following example configures two hosts `sarahconnor` and `johnconnor` to use the respective files `/tftpboot/sarahconnor.lst` and `/tftpboot/johnconnor.lst` as grub config.

```
# example dhcp config with invalid ethernet addresses
subnet 192.168.1.0 netmask 255.255.255.0 {
  group
  {
    filename '/tftpboot/pxegrub';
    # offer the host the here given name as host name
    option host-name = host-decl-name;
    option dhcp-parameter-request-list = concat(option dhcp-parameter-request-list,96);
    host sarahconnor
    {
      hardware ethernet 00:09:11:11:11:11;
      fixed-address 192.168.1.2;
      option configfile "/tftpboot/sarahconnor.lst";
    }
    host johnconnor
    {
      hardware ethernet 00:09:22:22:22:22;
      fixed-address 192.168.1.3;
      option configfile "/tftpboot/johnconnor.lst";
    }
  }
}
```

These grub config files are later dynamically overwritten for each boot by your application server’s “Master Control Program” (MCP).

The example above assumes the DHCP also running on the central Master Control Program (MCP) server. To use a DHCP server running on another host configure it with some grub/tftp redirection chains to in the end lead to the same files `/tftpboot/sarahconnor.lst` and `/tftpboot/johnconnor.lst` loaded from the MCP server.

- Force DHCP server to reread its configuration

```
$ kill -HUP $pid_of_dhcpd
```

### 3.1.4 Configure TFTP on central MCP machine

The MCP server is also acting as a TFTP server, so it has to be configured as such:

- Install a TFTP server

```
$ sudo apt-get install inetutils-inetd
$ sudo apt-get install atftpd
$ sudo chmod 777 /var/lib/tftpboot/
$ sudo ln -s /var/lib/tftpboot /tftpboot
```

- Create symlinks to point TFTP dir into Tapper working dir

The TFTP daemon only serves files from `/tftpboot`, as seen above in the DHCP config. To supply files from the Tapper working dir make the `/tftpboot` a symlink to the Tapper working dir.

```
$ ln -s /data/tapper/live/configs/tftpboot /tftpboot
```

When Tapper creates tftp files it works with absolute path names. Because the TFTP daemon interprets all absolute pathnames relative to its root dir we supply a `tftpboot` symlink inside the tftp root (which is also our Tapper working dir), so we can use the same absolute path name in both contexts (Tapper and TFTP):

```
$ ln -s /data/tapper/live/configs/tftpboot \
/data/tapper/live/configs/tftpboot/tftpboot
```

#### 3.1.5 Make the hosts known in the TestrunDB

```
$ tapper-testrun newhost --name=sarahconnor --active=1
$ tapper-testrun newhost --name=johnconnor --active=1
```

This makes the hosts generally available (active) for scheduling testruns by machine name. For scheduling hosts by more detailed machine features (cpu, memory, family, etc.) you need to add according key/value pairs in the `HostFeature` table.

#### 3.1.6 Optionally: enable temare to generate tests for this host

`Temare` is an utility that generates preconditions according to a test matrix of host/guest virtualization scenarios (but not yet shipped publicly).

For generating preconditions for a host, you can register the host in `temare`.

If you want tests scheduled for the new machine then follow these steps:

- Login as root on MCP server
- Set the `PYTHONPATH` to include the `temare` src directory

```
export PYTHONPATH=$PYTHONPATH:/opt/tapper/python/temare/src
```

- Add the host to temare hostlist

```
$ /opt/tapper/python/temare/temare hostadd $hostname \
                                           $memory \
                                           $cores \
                                           $bitness
```

---

### 3.1.7 Optionally: Web server for autoinstall

If you want to use autoinstall (i.e. kickstart, autoyast in contrast to image based installation with the Tapper installer) the suggested way to do it is Cobbler. If you do not want to use Cobbler, Tapper also supports kickstart more directly using the autoinstall precondition type. For this precondition you need a way to get the kickstart/autoyast to the test machine. The suggested way is a Web server, e.g Apache. Adapt your Apache config with the following: `cat <<EOF >>/etc/apache2/conf.d/tapper.conf Alias /autoinstall /data/tapper/live/configs/autoinstall <Directory /data/tapper/live/configs/autoinstall> Options +FollowSymLinks +Indexes ForceType text/plain order allow,deny allow from all </Directory> EOF`

- Add the Tapper ssh key to your image.

```
cat /home/tapper/.ssh/id_dsa.pub >> /root/.ssh/authorized_keys
```

(FIXME) Actually this does not belong into the host preparation but into a separate image preparation chapter which does not yet exist.

## 4 Test Protocol

In order to write test suites you need to understand the output protocol, which is TAP, the **Test Anything Protocol**.

The protocol is trivially to produce, you can do it with simple Shell `echos` or you can use TAP emitting toolchains, like practically all `Test::*` modules from the Perl world.

This chapter explains the protocol and the Tapper specific extensions, which are usually headers that can be transported inside TAP comments.

### 4.1 Test Anything Protocol (TAP)

#### 4.2 Tutorial

##### 4.2.1 Just plan and success

**Example:**

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

**Remarks:**

- 3 single tests planned
- the two first went good
- the last went wrong

#### 4.2.2 Succession numbers

**Example:**

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

**Remarks:**

- Missing test lines (eg. due to internal bummers) can be detected.

**Example with missing test:**

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

**Remarks:**

- Parsing will later say “ test 2 expected but got 3”

#### 4.2.3 Test descriptions

**Example:**

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

**Remarks:**

- Readability.

#### 4.2.4 Mark tests as TODO

**Example:**

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

**Remarks:**

- mark not yet working tests as "TODO"
- allows test-first development
- "ok" TODOs are recognized later

("unexpectedly succeeded")

=item \* We also use it to ignore known issues with still being able to find them later.

#### 4.2.5 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
```

##### Remarks:

- comment the TODO reason

#### 4.2.6 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
```

##### Remarks:

- mark tests when not really run (note it's set to "ok" anyway)
- keeps succession numbers in sync

#### 4.2.7 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'
```

##### Remarks:

- Unstructured details

### 4.2.8 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'
...
```

**Remarks:**

- Structured details
- 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

### 4.2.9 Meta information headers for reports

TAP allows comment lines, starting with #. We allow meta information transported inside those comment lines when declared with Tapper specific headers.

**Example:**

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

**Remarks:**

- we use diagnostics lines (“hot comments”)
- semantics only to our TAP applications

These are the headers that apply to the whole report:

```

# Tapper-suite-name:           -- suite name
# Tapper-suite-version:       -- suite version
# Tapper-machine-name:        -- machine/host name
# Tapper-machine-description:  -- more details to machine
# Tapper-reportername:        -- user name of the reporter
# Tapper-starttime-test-program: -- start time for complete test
                                (including guests)
# Tapper-endtime-test-program: -- end time for complete test
                                (including guests)
# Tapper-reportgroup-testrun:  -- associate this report with other
                                reports of same testrun_id
# Tapper-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)

```

There are more headers that apply to single sections of a report.

#### 4.2.10 Report sections

Standard TAP contains of exactly one block with one plan (eg., 1..5) and some TAP lines. In Tapper you can concatenate several such blocks at once. They are interpreted like different files, and are named *sections* in Tapper jargon.

The delimiter between such sections is the plan line. This requires the plan to come first for each section. See chapters “Explicit section markers with lazy plans” and “TAP archives” below for explicitly providing other TAP section delimiters.

Please remember: **Concatenating several sections into one big block of TAP is a Tapper extension. To interact with other TAP toolchains you should try to use “TAP archives” when submitting sections into Tapper.**

**Example:**

```

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

```

**Remarks:**

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

### 4.2.11 Meta information headers for report sections

These are the headers that apply to single sections:

```
# Tapper-explicit-section-start:  -- explicitly start a section now
                                -- instead of autorecognition

# Tapper-ram:                    -- memory
# Tapper-cpuinfo:                -- what CPU
# Tapper-uname:                  -- kernel information
# Tapper-osname:                 -- OS information
# Tapper-bios:                   -- BIOS information
# Tapper-flags:                  -- flags, usually linux kernel
# Tapper-changeset:              -- exact changeset of the currently
                                -- tested software or kernel

# Tapper-description:            -- more description of the currently
                                -- tested software or kernel,
                                -- e.g., if changeset is not enough

# Tapper-uptime:                 -- uptime, maybe the test run time
# Tapper-language-description:   -- for Software tests,
                                -- like "Perl 5.10", "Python 2.5"

# Tapper-reportcomment:          -- Freestyle comment

# Tapper-xen-version:            -- Xen version
# Tapper-xen-changeset:          -- particular Xen changeset
# Tapper-xen-dom0-kernel:        -- the kernel version of the dom0
# Tapper-xen-base-os-description: -- more verbose OS information
# Tapper-xen-guest-description:  -- description of a guest
# Tapper-xen-guest-test:         -- the started test program
# Tapper-xen-guest-start:        -- start time of test
# Tapper-xen-guest-flags:        -- flags used for starting the guest

# Tapper-kvm-module-version:     -- version of KVM kernel module
# Tapper-kvm-userspace-version:  -- version of KVM userland tools
# Tapper-kvm-kernel:             -- version of kernel
# Tapper-kvm-base-os-description: -- more verbose OS information
# Tapper-kvm-guest-description:  -- description of a guest
# Tapper-kvm-guest-test:         -- the started test program
# Tapper-kvm-guest-start:        -- start time of test
# Tapper-kvm-guest-flags:        -- flags used for starting the guest

# Tapper-simnow-version:         -- version of simnow
# Tapper-simnow-svn-version:     -- svn commit id of simnow
# Tapper-simnow-svn-repository:  -- used svn repository
# Tapper-simnow-device-interface-version: -- internal simnow device
```

```

                                interface version
# Tapper-simnow-bsd-file:        -- used BSD file (machine model)
# Tapper-simnow-image-file:     -- used OS image bottled in simnow
                                (usually similar to
                                Tapper-osname or
                                Tapper-xen-base-os-description or
                                Tapper-kvm-base-os-description)

```

#### 4.2.12 Meta information structure summary

There are groups of reports (e.g. for virtualization scenarios), optionally identified by a testrun ID or by an arbitrary ID. Every report has an ID and a set of meta information. A report consists of sections, which can each have section specific set of meta information.

The resulting meta information hierarchy looks like this.

- Reportgroup
  - testrun reportgroup ID
  - arbitrary reportgroup ID
- Report
  - report ID
  - Tapper-suite-name
  - Tapper-suite-version
  - Tapper-machine-name
  - Tapper-machine-description
  - Tapper-reportername
  - Tapper-starttime-test-program
  - Tapper-endtime-test-program
  - Tapper-reportgroup-testrun
  - Tapper-reportgroup-arbitrary
- \* Section
  - Tapper-explicit-section-start
  - Tapper-ram
  - Tapper-cpuinfo
  - Tapper-uname
  - Tapper-osname
  - Tapper-bios
  - Tapper-flags
  - Tapper-changeset
  - Tapper-description
  - Tapper-uptime
  - Tapper-language-description
  - Tapper-reportcomment
  - Tapper-xen-version
  - Tapper-xen-changeset
  - Tapper-xen-dom0-kernel
  - Tapper-xen-base-os-description

- Tapper-xen-guest-description
- Tapper-xen-guest-test
- Tapper-xen-guest-start
- Tapper-xen-guest-flags
- Tapper-kvm-module-version
- Tapper-kvm-userspace-version
- Tapper-kvm-kernel
- Tapper-kvm-base-os-description
- Tapper-kvm-guest-description
- Tapper-kvm-guest-test
- Tapper-kvm-guest-start
- Tapper-kvm-guest-flags
- Tapper-simnow-version
- Tapper-simnow-svn-version
- Tapper-simnow-svn-repository
- Tapper-simnow-device-interface-version
- Tapper-simnow-bsd-file
- Tapper-simnow-image-file

#### 4.2.13 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 Tapper 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 Tapper header `Tapper-explicit-section-start` to explicitly start a section. Everything until the next header `Tapper-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 `Tapper-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:**

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

Please note again: **The sectioning in general and this auxiliary header for marking sections is a Tapper extension, not standard TAP. An alternative way better than fiddling with this sectioning is to produce TAP archives and submit them instead. See chapter “TAP Archives”.**

#### 4.2.14 Developing with TAP

TAP consuming is provided via the `Test::Harness` aka. `TAP::Parser` Perl toolchain. The frontend utility to execute TAP emitting tests and evaluate statistics is `prove`.

```
$ 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
```

#### Remarks:

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

It helps to not rely on Tapper extensions (like report sections) when using the `prove` command.

#### 4.2.15 TAP tips

- TAP is easy to produce but using it **usefully** can be a challenge.
- Use **invariable** test descriptions.
- Put meta information in diagnostics lines, **not** into test descriptions.
- Use the description after `# TODO/SKIP`.
- Cheat visible (or: don't cheat invisible).
- Really use `# TODO/SKIP`.

These tips keep later TAP evaluation consistent.

### 4.3 Particular use-cases

#### 4.3.1 Report Groups

**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

```
# Tapper-reportgroup-testrun: 1234
```

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

**Report grouping by arbitrary identifier** If the grouping id is not a testrun id, e.g., because you have set up a Xen environment without the Tapper 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:

```
TAPPER_REPORT_GROUP='date|md5sum|awk '{print $1}''
```

- use the value:

```
# Tapper-reportgroup-arbitrary: $TAPPER_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 `TAPPER_REPORT_GROUP`.

### 4.4 TAP Archives

Some TAP emitting toolchains allow the generation of `.tar.gz` files containing TAP, so called *TAP archives*. E.g., via `prove`:

```
$ prove -a /tmp/myresults.tgz t/
```

You can later submit such TAP archive files to the Tapper reports receiver the same way as you report raw TAP.

### 4.5 Reporting TAP to the reports receiver

The Tapper reports receiver is a daemon that listens on a port and slurps in everything between the open and close of a connection to it. Therefore you can use `netcat` to report TAP.

Remember that using `netcat` in turn can be a mess, there are several flavours with different options which are also changing their behaviour over time. So to be sure, you better do your own socket communication with Perl or Python: open socket, print to socket, close socket, done. We just keep with `netcat` for illustrating the examples.

---

### 4.5.1 Submitting raw TAP

Simply submit all TAP directly into the socket of the reports receiver:

```
$ ./my_tap_emitting_test_suite | netcat tapper_server 7357
```

### 4.5.2 Submitting TAP Archives

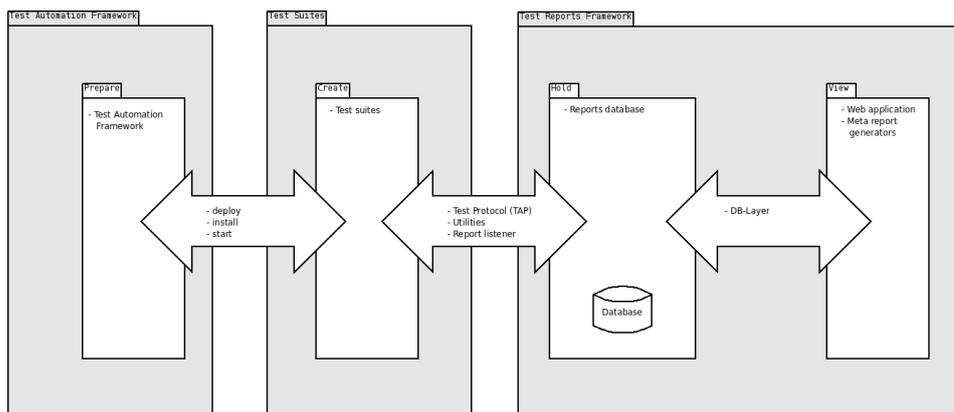
You submit the content of a `.tar.gz` file in the same way you submit raw TAP, via the same API. The receiver recognizes the `.tar.gz` contenttype by itself.

```
$ prove -a /tmp/myresults.tgz t/  
$ cat /tmp/myresults.tgz | netcat tapper_server 7357
```

## 5 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:



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

```
TAPPER_REPORT_SERVER  
TAPPER_REPORT_PORT
```

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

### 5.1 Available test suite wrappers

Originally we have a lot of direct wrappers available but haven't them all published as open source. For OS testing the most important wrapper which is also publicly available is `tapper-testsuite-autotest` aka. `Tapper-Testsuite-AutoTest`. You should look at that.

### 5.1.1 tapper-testsuite-autotest

A suite that wraps the *autotest* client with the export of TAP and sends the resulting TAP archives to Tapper server.

That is the primary testsuite wrapper for OS testing.

## 5.2 Environment variables

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

- `TAPPER_TESTRUN` Currently active Testrun ID.
- `TAPPER_SERVER` The controlling automation Server that initiated this testrun.
- `TAPPER_REPORT_SERVER` The target server to which the tests should report their results in TAP.
- `TAPPER_REPORT_PORT` The target port to which the tests should report their results in TAP. Complements `TAPPER_REPORT_SERVER`.
- `TAPPER_REPORT_API_PORT` The port on which the more sophisticated Remote Reports API is available. It's running on the same host as `TAPPER_REPORT_SERVER`.
- `TAPPER_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.)
- `TAPPER_GUEST_NUMBER` Virtualisation guests are ordered, this is the guest number or 0 if not a guest.
- `TAPPER_NTP_SERVER` The server where to request NTP dates from.

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

## 6 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.

## 6.1 SYNOPSIS

### 6.2 Layers of preconditions

### 6.3 Precondition repository

### 6.4 Overview Precondition Types

### 6.5 Details Precondition Types

### 6.6 Macro Preconditions

### 6.7 Testplans

### 6.8 Precondition Producers

- 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 testrun is executed and results are reported

### 6.9 Layers of preconditions

There are “normal preconditions” and “macro preconditions”.

#### 6.9.1 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 formatted 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.

Please note: **Normal preconditions are usually not what you want.** It's the low level mechanism. Its advantage is in reusing the preconditions by referring to IDs and creating trees of preconditions. This reuse is usually too complex. **What you typically want are Macro Preconditions.**

#### 6.9.2 Macro preconditions

There is another mechanism on top of normal preconditions: **Macro Preconditions**. These allow to bundle **multiple** preconditions at once into a common use-case.

A *macro precondition* is evaluated when the testrun is added via the cmdline utils (or the web app, both use the same underlying layer). The result are “normal preconditions” which are inserted into the DB everytime together with the testrun, so there is no reuse of preconditions and preconditions are always a list, no tree. Anyhow, they are much easier to handle.

*Macro preconditions* are template files which should be archived in the precondition repository, as only the finally resulting preconditions are stored in the database.

### 6.9.3 Testplans

Testplans are a variant of macro preconditions to control multiple testruns and queue them to particular hosts and queues. They are mostly useful to provide end-users with complex but polished frontend use-cases where they can define values to prepared parameters.

### 6.10 Precondition repository

Macro preconditions can be stored in

```
/data/tapper/live/repository/macropreconditions/
```

### 6.11 Overview Precondition Types

There are two variants of preconditions: *Action preconditions* and *Highlevel preconditions*. Action preconditions describe single actions, like “copy a file” or “execute a program”. Highlevel preconditions can contain other (action) preconditions and are used for instance for virtualization install scenarios where hosts and guests are described.

**Please note the wording:** A precondition is the particular YAML block with all the details (think of an object instance). Such a block is of a “precondition type” which defines its allowed structure (think of a class).

#### 6.11.1 Overview: Action precondition types

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.
- `cobbler`  
Use Cobbler to install the test system.

### 6.11.2 Overview: Highlevel precondition types

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).

## 6.12 Details Precondition Types

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.

### 6.12.1 installer\_stop

stop run after system installer

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

### 6.12.2 grub

overwrite automatically generated grub config

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

- Note: multiple lines in the grub file have to be given as one line separated by “\n” (literally a backslash and the letter 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 |, a newline and some indentation

### 6.12.3 package

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

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

### 6.12.4 copyfile

a file that just needs to be scp or copied:

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

- 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.

### 6.12.5 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"
```

### 6.12.6 image

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

```
---
precondition_type: image
mount: /
partition: testing
image: /data/tapper/live/repository/images/rhel-5.2-rc2-32bit.tgz
```

- 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 formatted `ext3` (only when image is given) and mounted to mount afterwards (this is why image exists at all, `copyfile` does not provide this)
- “image”: absolute or relative path (relative to `/data/tapper/live/repository/images/`)  
If not given, then it re-uses the partition without formatting/unpacking it.
- partition: Can be `/dev/XXX` or LABEL or UUID.

### 6.12.7 repository

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

- `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 `/`).

### 6.12.8 type: prc

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

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

```
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
```

- 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 `TAPPER_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.

### 6.12.9 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 following lesson:
```

### 6.12.10 type: hint

Such a precondition provides hints where normal behaviour needs to be changed. It contains any hash keys needed for the special handling. The special handling itself is done in the MCP and needs to be prepared for what you specify here.

We currently use it to handle SimNow testing.

```
precondition_type: hint
simnow: 1
script: family10_sles10_xen.simnow
```

### 6.12.11 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
```

### 6.12.12 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
```

### 6.12.13 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/tapper/.../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
```

### 6.12.14 type: testprogram

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

Testprograms allow a key **chdir** which can either contain a particular directory where to chdir before testprogram execution or the value **AUTO** (uppercase) which means to chdir to the dirname of the program.

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

**6.12.15 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 absolute path or relative to `/data/tapper/.../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
```

**6.12.16 type: cobbler**

Install a system using Cobbler. The profile denotes a profile exactly as known by Cobbler. The timeout allows you to limit the time used for installation.

```
precondition_type: cobbler
profile: ubuntu-12.04-64
timeout: 1800
```

**6.12.17 type: virt**

A virtualization environment.

(The linebreaks with `\` are not part of the actual file, but only for this document.)

```
precondition_type: virt
name: automatically generated Xen test
host:
  preconditions:
    - precondition_type: package
      filename: /data/tapper/live/repository/packages/xen/builds/\
                x86_64/xen-3.3-testing/\
                xen-3.3-testing.2009-03-20.18614_f54cf790ffc7.x86_64.tgz
    - precondition_type: package
      filename: /data/tapper/live/repository/packages/tapperutils/\
                sles10/xen_installer_suse.tar.gz
    - precondition_type: exec
      filename: /bin/xen_installer_suse.pl
  root:
    precondition_type: image
    partition: testing
    image: suse/suse_sles10_64b_smp_raw.tar.gz
    mount: /
    arch: linux64
  testprogram:
    execname: /opt/tapper/bin/tapper_testsuite_dom0_meta.sh
    timeout_testprogram: 10800
  guests:
    - config:
      precondition_type: copyfile
```

```
protocol: nfs
name: tapper:/data/tapper/live/repository/configs/\
      xen/001-sandschaki-1237993266.svm
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/tapper/bin/py_ltp
timeout_after_testprogram: 10800
```

- 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 tapper toolchain)

### 6.12.18 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

## 6.13 Macro Preconditions

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

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

In most cases “normal 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 “normal 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

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

These `# tapper-*` headers are also used in web frontend to render forms out of it and submit testruns from there.

The values for the placeholders can be filled via such a command line:

```
$ tapper-testrun new [all usual options] \
  --macroprecond=FILENAME \
  -Did=value1 \
  -Dkernel=2.6.37
```

The FILENAME is a complete filename with absolute path.

There is no restriction on TemplateTsubstitution. 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%]
```

### 6.13.1 A real live example - kernel boot test

- Macroprecondition

```
# tapper-mandatory-fields: kernel_version
# tapper-optional-fields: kernelpkg
---
precondition_type: image
arch: linux64
image: suse/suse_sles10_64b_smp_raw.tar.gz
mount: /
partition: testing
---
precondition_type: copyfile
```

```
name: /data/tapper/live/repository/testprograms/uname_tap/uname_tap.sh
dest: /bin/
protocol: local
---
precondition_type: copyfile
name: /data/tapper/live/repository/packages/tapperutils/kernel/gen_initrd.sh
dest: /bin/
protocol: local
---
[% kernelpkg = BLOCK %]\
[% IF kernelpkg %]\
[% kernelpkg %]\
[%ELSE%]kernel/linux-[% kernel_version %].tar.gz[% END %]\
[% END %]
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 "# Tapper-Suite-Name: Kernel-Boot"
echo "# Tapper-Suite-Version: 1.00"
echo "# Tapper-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:

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

or with some more information (owner, topic):

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

## 6.14 Testplans

This section describes testplan files as they are stored in `/data/tapper/live/testplans/`.

A testplan is very similar to a macro precondition but can additionally bundle several testruns and specify required hostname and used queue inside. This and the possibility of using Template-Toolkit (see Macro Preconditions) inside makes it most useful for frontend usage: You can take all parameters, especially queue+hosts from commandline via `-Dhosts=affe,zomtec,birne`, split them via Template Toolkit and loop to create multiple testruns.

To tell it's one test out of a set, we use the type `multitest`, like this:

```
---
type: multitest
description:
  shortname: my-testplan
  topic: my-testplan
  queue: AdHoc
  requested_hosts_all:
  - affe
  - zomtec
preconditions:
  - ...
  - ...
```

The preconditions are then as already described in earlier chapters.

It additionally allows to embed self-documentation which can be viewed with the command `tapper-testrun newtestplan --guide --file my-testplan-file`.

To take parameters and create multiple testruns, you could use it like this:

```
### This is an example testplan
### Allowed params:
```

```
### - -Dqueue=QUEUE
### - -Dtests=TEST1,TEST2,TEST3
### - -Dmachines=HOST1,HOST2
[%# define default values %]
[%- IF queue == '' %] [% queue = 'AdHoc' %] [% END -%]
[%- IF tests == '' %] [% tests = 'hackbench' %] [% END -%]
[%- IF machines == '' %] [% machines = 'arges' %] [% END -%]
[% AllTests = tests.split(',') %]
[% AllDistros = distros.split(',') %]
[% AllMachines = machines.split(',') %]
[%- FOREACH machine = AllMachines %]
[%- FOREACH test = AllTests %]
---
type: multitest
description:
  shortname: [% test %]
  topic: Topic-[% AllTests.join('-') %]
  queue: [% queue %]
  requested_hosts_all:
  - [% machine %]
  preconditions:
  - ...
  -
    precondition_type: testprogram
    program: /opt/tapper/bin/tapper-testsuite-autotest
    parameters:
    - --test
    - [% test %]
  - ...
```

From a command line this can be used like below.

Show self-doc:

```
$ tapper-testrun newtestplan --guide --file my-testplan
```

```
This is an example testplan
Allowed params:
- -Dqueue=QUEUE
- -Dtests=TEST1,TEST2,TEST3
- -Dmachines=HOST1,HOST2
```

Start a new testplan consisting of potentially many testruns:

```
$ tapper-testrun newtestplan --verbose \
    --file my-testplan \
    -Dqueue=TrackWorkload \
    -Dtests=hackbench,dbench
```

```
Plan created
  id:
  241
  url: http://tapper/tapper/testplan/id/241
  path: topic/kernel/misc/my-testplan
  file: /data/tapper/topic/kernel/misc/my-testplan
```

Whenever you want to provide a polished use-case to an end-user consider writing a testplan file. You add all the complexity, details and documentation into the file and provide your user with just this file. He/she can then specify the prepared parameters for his/her particular needs, like the tests to run.

We additionally use Testplans to bi-directional map from such testplans to a task in our planning software where we simply assume a mapping of the task hierarchy to a subdirectory hierarchy. Based on this mapping the planning software can specify when to execute such a testplan according to the project plan.

A testplan then typically contains several testruns: some for the known-good sanity tests without a changed feature and some for the actual tested feature.

## 6.15 Precondition Producers

Sometimes, parameters for preconditions shall be defined when 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*.

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 precondition 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.

### 6.15.1 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.

### 6.15.2 Producer API

Producer are modules loaded into the scheduler. Thus they need to be written in Perl and reside inside the `Tapper::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 `Tapper::MCP::Scheduler::PreconditionProducer`. Producers shall 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 canceled. In this case the other hash keys are not evaluated.

### 6.15.3 Existing producers

Currently the following producers exist:

- `DummyProducer.pm`  
Dummy producer for testing.
- `Kernel.pm`  
Produces preconditions for kernel tests.
- `NewestPackage.pm`  
Produces a package precondition that installs the newest package from a given directory.
- `SimnowKernel.pm`  
Produces preconditions for simnow kernel tests.
- `Temare.pm`  
Wraps the existing temare producer utility.

### 6.15.4 Example: “Kernel” precondition producer

The kernel producer returns a package precondition that contains the latest kernel package from the kernel package path. Furthermore, it returns an exec precondition that triggers the creation of an `initrd` file for the kernel.

Valid options:

- `arch`  
May be `x86_64` or `i686`. The latest kernel package from the associated path are used.
- `version`  
Only use kernel packages that contain the given version string
- `stable`  
Use stable kernels when true

**Lazy precondition** The lazy precondition, pointing to the “Kernel” precondition producer:

```
precondition_type: produce
producer: Kernel
arch: i686
version: 2.6.32
stable: 1
```

---

**Resulting preconditions** The resulting preconditions may look like this:

```
---
precondition_type: package
filename: kernel/stable/i686/kernel-2.6.32-rc1.tar.gz
---
precondition_type: exec
filename: /bin/gen_initrd.sh
options:
  - 2.6.32-rc1
```

## 7 Command line interface

### 7.1 Commandline Synopsis

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

### 7.2 Scheduling

#### 7.2.1 Create new queue and new host and bind both together

- Show existing queues with priorities

```
$ tapper-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*

```
$ tapper-testrun newqueue --name=oprofile \
                        --priority=200
12
```

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

```
$ tapper-testrun newhost --name=bullock \
                        --queue=oprofile
10
```

- Show existing hosts

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

```
$ tapper-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.

```
$ tapper-testrun updatehost --id=10 --active
10 | bullock | active | free | oprofile
```

- Again, show existing hosts

Host *bullock* is now activated.

```
$ tapper-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.

## 7.2.2 Change queue priority

- List existing queues

```
$ tapper-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

```

$ tapper-testrun updatequeue --name=oprofile \
                             --priority=1000
12

```

- Again, list existing queues

```

$ tapper-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.

### 7.2.3 requested features

Hosts for testruns can be chosen based on requested features. Supported features are:

- hostname
- mem
- vendor
- family
- model
- stepping
- revision
- socket
- cores
- clock
- l2cache
- l3cache

### 7.2.4 Cancel current testrun on host

Freeing a host need the config for the currently running testrun. Thus, the command is only tested on bancroft and may not work on other machines.

```
$ tapper-testrun freehost \  
    --name=bullock\  
    --desc='I need this host right now'
```

## 8 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.

### 8.1 Usage

The main URL is typically something like

```
http://tapper/tapper
```

### 8.2 Understanding Tapper Details

#### 8.2.1 Part 1 Overview

- Go to <http://tapper/tapper/reports>
- Click “Last weeks test reports”, aka. <http://tapper/tapper/reports/date/7>
- Below day “Wed Oct 7, 2009” find the line

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

- 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	PASS
CTCS	satyr:eriador	FAIL

- Click on the ID link “20856” aka. <http://tapper/tapper/reports/id/20856>

### 8.2.2 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. <http://tapper/tapper/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. <http://tapper/tapper/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.

### 8.2.3 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. <http://tapper/tapper/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_32bpaе_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.

---

## 9 Reports API

### 9.1 Overview

There runs yet another daemon, the so called `Tapper::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 `Tapper::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).

### 9.2 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 *Client Utility `tapper-api`* 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.

#### 9.2.1 upload aka. 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.

#### upload Synopsis

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

#### 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.

**upload Payload** The raw content of the file to upload.

**upload 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
```

### 9.2.2 download - download a file which is attached to a report

This api command lets you download files, aka. attachments, from reports.

#### download Synopsis

```
#! upload REPORTID FILENAME
```

There is no other payload necessary here, just this single line.

#### download Parameters

- **REPORTID**  
The id of the report to which the file is assigned
- **FILENAME**  
The name of the file as it was specified on upload

**download Example usage** Just echo the first api-command line and redirect the answer into a file.

```
$ ( echo "#! download 552 xyz.tmp" ) | netcat -w1 bascha 7358 > xyz.tmp
```

### 9.2.3 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.

#### mason Synopsis

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

#### mason 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.

**mason 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 DPath* for details about this.

**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
```

### 9.3 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*.

### 9.3.1 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.

**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' } }
```

### The data structure

### 9.3.2 Data Filter

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/>).

### Data-DPath expressions

### 9.3.3 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,
```

## 9.4 Client Utility `tapper-api`

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

---

### 9.4.1 help

You can acquire a help page to each sub command:

```
$ /opt/tapper/perl/bin/tapper-api help upload
```

prints

```
tapper-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/octet-stream' for binaries
```

### 9.4.2 upload

Use it from the Tapper path, like:

```
$ /opt/tapper/perl/bin/tapper-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 | /opt/tapper/perl/bin/tapper-api upload \
  --file=- \
  --filename dmesg \
  --reportid=301
```

### 9.4.3 mason

TODO

## 10 Complete Use Cases

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

### 10.1 Automatic Xen testing

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

The following mainly applies to **manually** assigning Xen tests. In the OSRC we use *temare* (not yet published) to automatically create the here described steps.

#### 10.1.1 Paths

- Host **tapper**: `/data/tapper/live/`
- Host **osko**: `/export/image_files/official_testing/`

### 10.1.2 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/images/testing/raw/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 `tapper:/data/tapper/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 section for this guest is:

```
root:
  precondition_type: copyfile
  name: osko:/export/images/testing/raw/redhat_rhel5u2_64b_smp_up_small_raw.img
  protocol: nfs
  dest: /xen/images/
  mountfile: /xen/images/redhat_rhel5u2_64b_smp_up_small_raw.img
  mountpartition: p1
```

### 10.1.3 PRC configuration

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

**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/tapper/packages/mhentsc3/001.svm
  protocol: local
  dest: /xen/images/
  filename: /xen/images/001.svm

```

**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/tapper/live/repository/packages/"`. Since `"tapper:/data/tapper/"` is mounted to `"/data/tapper/"` in the install system, this directory can be accessed at `"tapper:/data/tapper/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 Tapper waiting forever if your test doesn't send finish messages. The resulting testprogram section looks like this:

```

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

```

#### 10.1.4 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.

### 10.1.5 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: tapper-testsuite-system.tar.gz
    path: mhentsc3/
    timeout_after_testprogram: 3600
    execname: /home/tapper/x86_64/bin/tapper_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/images/testing/raw/redhat_rhel5u2_64b_smp_up_small_raw.img
  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/tapper/packages/mhentsc3/001.svm
  protocol: local
  dest: /xen/images/
  filename: /xen/images/001.svm
testprogram:
  precondition_type: copyfile
  name: /usr/share/tapper/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/

```

### 10.1.6 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 of the root partition (including parenthesis). 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
```

### 10.1.7 Order Testrun

To order your test run with the previously defined preconditions you need to stuff them into the database. Fortunately there are commandline tools to help you with this job. They can be found at `/opt/tapper/perl/perl/current/bin`. In our production environment the server for Tapper is `tapper`. Log in to this server (as user `tapper`). Make sure that `/opt/tapper/perl/perl/current/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 `tapper-testrun newprecondition` with the appropriate options, e.g. in our example:

```
$ tapper-testrun newprecondition --shortname=grub \
                                --condition_file=/tmp/grub.yml

$ tapper-testrun newprecondition --shortname=xen \
                                --condition_file=/tmp/xen.yml
```

`tapper-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 `tapper-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. Add `--owner` with an appropriate user if you don't want the default `tapper`. The resulting call looks like this:

```
tapper-testrun new --hostname=bullock \
                  --precondition=4 \
                  --precondition=5 \
                  --test_program=whatever \
                  --owner=mhentsc3
```

Please note: There is a more central approach to describe all needed preconditions at once, see [Macro Preconditions](#) and [A real live example - kernel boot test](#).

When the requested testrun is scheduled Tapper will setup the system you requested and execute your defined testrun. When everything went well, you'll see test output soon after. For more information on what is going on with Tapper, see `/var/log/tapper-debug`.

## Index

- A**
- A real live example - kernel boot test ..... 40
  - Adding a new host into automation ..... 13
  - Automatic Xen testing.....56
  - Automation.....9
  - Available test suite wrappers .....28
- C**
- Cancel current testrun on host ..... 49
  - Change queue priority ..... 47
  - Choose an image for Dom0 and images for each guest .....57
  - Client Utility tapper-api ..... 55
  - Command line interface .....46
  - Commandline Synopsis ..... 46
  - Comment TODO tests with reason.....20
  - Complete Use Cases ..... 56
  - Configure TFTP on central MCP machine . 16
  - copyfile ..... 33
  - Correlation to autotest.kernel.org ..... 11
  - Create new queue and new host and bind both together ..... 46
- D**
- Data Filter ..... 55
  - Data-DPath expressions ..... 55
  - Details Precondition Types.....30, 32
  - Developing with TAP .....26
  - Diagnostics ..... 20
  - download - download a file which is attached to a report ..... 53
  - download Example usage ..... 53
  - download Parameters ..... 53
  - download Synopsis.....53
- E**
- Environment variables ..... 29
  - Example usage .....54
  - Example: “Kernel” precondition producer .. 45
  - Existing producers .....45
  - Explicit section markers with lazy plans....25
- F**
- fstab.....34
- G**
- General precondition keys “mountfile” .....39
  - Grub ..... 60
  - grub ..... 32
  - Guest Start Configuration ..... 57
- H**
- help ..... 56
- I**
- image .....34
  - Infrastructure .....9
  - installer stop.....32
- J**
- Just plan and success ..... 18
- L**
- Layers of preconditions ..... 30
  - Lazy precondition ..... 44, 45
- M**
- Machine specific configuration ..... 14
  - Macro Preconditions ..... 30, 40
  - Macro preconditions ..... 30
  - Make machine PXE boot aware.....16
  - Make machine remote hard resetable.....14
  - Make the hosts known in the TestrunDB...17
  - Many use-cases ..... 10
  - Mark tests as SKIP (with reason) .....20
  - Mark tests as TODO ..... 19
  - mason ..... 56
  - mason - Render templates with embedded query language .....53
  - mason Parameters ..... 53
  - mason Payload ..... 54
  - mason Synopsis .....53
  - Master Control Program (MCP) ..... 12
  - Meta information headers for report sections23
  - Meta information headers for reports.....21
  - Meta information structure summary ..... 24
- N**
- Normal preconditions ..... 30
- O**
- Optimizations ..... 55
  - Optionally: enable temare to generate tests for this host.....17

Optionally: Web server for autoinstall . . . . .	18	Succession numbers . . . . .	19
Order Testrun . . . . .	61	Support for writing tests . . . . .	10
Overview . . . . .	9, 52	SYNOPSIS . . . . .	30
Overview Precondition Types . . . . .	30, 31	Synopsis . . . . .	11
Overview: Action precondition types . . . . .	31		
Overview: Highlevel precondition types . . . . .	32		
		<b>T</b>	
		TAP Archives . . . . .	27
<b>P</b>		TAP archives . . . . .	13
package . . . . .	33	TAP tips . . . . .	26
Parameters . . . . .	52	Tapper infrastructure . . . . .	11
Part 1 Overview . . . . .	49	tapper-testsuite-autotest . . . . .	29
Part 2 Details . . . . .	50	Technical Infrastructure . . . . .	13
Part 3 Testrun . . . . .	51	Technology . . . . .	11
Particular use-cases . . . . .	27	Test Anything Protocol (TAP) . . . . .	13, 18
Paths . . . . .	56	Test descriptions . . . . .	19
PRC configuration . . . . .	57	Test Protocol . . . . .	18
Precondition Producers . . . . .	30, 44	Test Suite Wrappers . . . . .	28
Precondition repository . . . . .	30, 31	Testplan Support . . . . .	10
Preconditions . . . . .	13, 29, 58	Testplans . . . . .	30, 31, 42
Producer API . . . . .	44	Testrun . . . . .	13
Program Run Control (PRC) . . . . .	12	Testrun DB . . . . .	13
		Testsuite Configuration . . . . .	58
		The data structure . . . . .	55
		Tutorial . . . . .	18
<b>Q</b>		type: autoinstall . . . . .	37, 38
Query language DPath . . . . .	54	type: cobbler . . . . .	38
quote subtleties . . . . .	36	type: exec . . . . .	35
		type: hint . . . . .	36
		type: prc . . . . .	35
		type: reboot . . . . .	37
		type: testprogram . . . . .	37
		type: virt . . . . .	38
		<b>U</b>	
<b>R</b>		Understanding Tapper Details . . . . .	49
Raw API Commands . . . . .	52	upload . . . . .	56
Report . . . . .	13	upload aka. attach a file to a report . . . . .	52
Report grouping by arbitrary idenitifer . . . . .	27	upload Example usage . . . . .	53
Report grouping by same testrun . . . . .	27	upload Payload . . . . .	52
Report Groups . . . . .	27	upload Synopsis . . . . .	52
Report sections . . . . .	22	Usage . . . . .	49
Reporting TAP to the reports receiver . . . . .	27		
Reports API . . . . .	12, 52		
Reports DB . . . . .	13		
Reports Filter (SQL-Abstract) . . . . .	55		
Reports Receiver . . . . .	12		
repository . . . . .	34		
requested features . . . . .	48		
Result Evaluation . . . . .	10		
Resulting preconditions . . . . .	46		
Resulting YAML config . . . . .	59		
		<b>V</b>	
		Vocabulary . . . . .	12
<b>S</b>			
Scheduling . . . . .	9, 46		
SQL::Abstract expressions . . . . .	55		
Submitting raw TAP . . . . .	28	<b>W</b>	
Submitting TAP Archives . . . . .	28	Web Application . . . . .	10
		Web User Interface . . . . .	12, 49

**Y**

YAML Diagnostics.....21