PAS Manual

Release 0.1a

Jonathan Stoppani

February 04, 2011

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 About

The PAS package provides a command line utility and a measuring framework for network analysis for the Parallel Object Programming protocol.

The tool allows to start POP-C++ nodes, run measure cases, capture the traffic and produce a complete report of the exchanged TCP packets between the peers.

By default, the test infrastructure is set up using vagrant and some virtual machines automatically configured using chef, but it is possible to use any remote machine with ssh access and some configuration work (involving mainly installing the needed dependencies and creating some shared folders).

Although this tool was developed especially for the C++ implementation of the POP model, it was conceived to make as less assumptions as possible about the underlying implementation and could easily be adapted to measure POP objects offered by other implementations such, for example, as POP-Java

Further information about the POP-C++ project and the POP model can be found on the project's homepage[1].

1.2 Requirements

The requirements necessary to install and run the `pas` utility are the following:

- Python >= 2.6 – http://python.org/;
- `vagrant` and VirtualBox to run the virtual machines – http://vagrantup.com/;
- A webkit based browser (Safari, Google Chrome,...) to display the reports.

The following libraries and utilities are optional but add some additional features:

- `xmllint` used to reformat the XML documents produced by the measures;
- `tidy` used to cleanup the HTML output of the reports (note that the absence of this utility will add more entries to the list of unsupported browsers).

The Python packages that will automatically be installed by setuptools are the following ones:

- `fabric` to dispatch commands to remote machines through ssh – http://fabfile.org/;
- `pygments` to highlight the source code and the decoded transactions – http://pygments.org/;
- `lxml` to transform the different measure xml files – http://codespeak.net/lxml/.

[1] http://gridgroup.hefr.ch/popc/

1.3 Installation

Until a stable release is packaged and uploaded to the cheese shop, the latest development snapshot can be installed directly from the github hosted sources using `easy_install` or `pip`:

```
$ pip install https://github.com/GaretJax/pop-analysis-suite/tarball/master
```

Note: Often, depending on the target system, the installation of a python package requires `root` privileges. If the previous command fails, retry to run it with `sudo`.

To check if the package was correctly installed, run the the `pas` command on the command line:

```
$ pas
```

You should obtain an incorrect usage error similar to the following:

```
usage: pas [-h] [--version] [-v] [-q] [--settings SETTINGS_MODULE]
           {authorize,execute,jobmgr,compile,init,measure} ...
pas: error: too few arguments
```

1.3.1 Install from source

It is possible to install the PAS package directly from source. The following commands should get you started:

```
$ wget --no-check-certificate https://github.com/GaretJax/pop-analysis-suite/tarball/master
$ tar -xzf GaretJax-pop-analysis-suite-*.tar.gz
$ cd GaretJax-pop-analysis-suite-*
$ python setup.py install
```

1.3.2 Setuptools

To install the PAS package, the setuptools package is required (for both source or remote installation modes). Albeit coming preinstalled on all major unix based operating systems you may need to install it.

You can obtain further information about `setuptools` either on its pypi project page[2] or on its official homepage[3].

1.4 Development

The whole development of the PAS project happens on github, you can find the source repository at the following address: https://github.com/GaretJax/pop-analysis-suite/

Further development/contribution directives will be added as soon as some interest is manifested by any hacker willing to participate in the development process.

1.5 Structure of this manual

This manual is conceived to offer an incremental approach to all feature which the `pas` has to offer. It is structured in three main parts:

[2] http://pypi.python.org/pypi/setuptools
[3] http://peak.telecommunity.com/DevCenter/setuptools

1. The first part describes and documents the common usage of the tool, the assumptions it mades about different aspects such as VM setups, object communications, file locations and so on.

   Once read and understood this first part, a user should be able to complete a full measure cycle with a custom developed measure case and generate a report to further analyze.

2. The second part dives in the internals of the `pas` libraries and documents topics such as command customization, custom type parsing or complex measurement setups, useful if a user want to completely adapt the offered functionalities to his needs.

3. The third part contains reference documentation useful for both a basic or an advanced usage. There are references for all built-in commands, for the different settings directives or for more advanced topics such as the internal APIs.

This three parts are completed with this introduction, a glossary of common terms and an alphabetical content index.

1.5.1 Building from source

The present document is written using Sphinx[4] and it can either be read online[5] thanks to readthedocs.org[6] or built locally using the `sphinx` tool into various different formats.

To create a local build, make sure to have the `sphinx` package installed and run the following commands:

```
$ git clone https://github.com/GaretJax/pop-analysis-suite/
$ cd pop-analysis-suite/docs
$ make html  # or any other format; run make without arguments to find out
             # the supported ones
```

The documentation builds will then be placed in the `_build/<format>` subdirectory.

[4] http://sphinx.pocoo.org/
[5] http://readthedocs.org/docs/pas/
[6] http://readthedocs.org

CHAPTER

TWO

USAGE DOCUMENTATION

2.1 Quick start

The steps contained in this short guide should get you up and running in as few steps as possible. For a full featured environment and for an in-dept description of all the functionalities of the `pas` packages, refer to the full documentation.

The following instructions assume that you have already installed the `pas` packages along with all its dependencies, if this is not the case, refer to the *Installation* instructions.

You don't need to understand completely what's going on in the following steps, but the *Execution model* and the *Commands reference* are a great resource to help you to grasp the details.

2.1.1 Step 1: Setup the test environment

To run a measure, a test environment has to be setup. Fortunately this can easily be done using the `init` command.

The init command tries to create a new environment in the current directory or using the path provided on the command line and will only succeed if it is empty. Use it in the following way:

```
$ pas init <path-to-the-env>
Attempting to create a new environment in '<path-to-the-env>'...
Done.
```

If the command executed successfully, a newly created directory will be present at the path you specified. Let's take a look at its content:

```
$ ls -l <path-to-the-env>
total 8
-rw-r--r--   1 garetjax  staff  878 Jan 25 14:52 Vagrantfile
drwxr-xr-x   4 garetjax  staff  136 Jan  3 21:30 cases
drwxr-xr-x   6 garetjax  staff  204 Jan  4 00:00 conf
drwxr-xr-x   4 garetjax  staff  136 Jan 12 21:51 contrib
drwxr-xr-x   2 garetjax  staff   68 Jan  3 14:29 reports
-rw-r--r--   1 garetjax  staff    0 Jan 25 14:52 settings.py
```

The entries in the listing have the following roles:

- `Vagrantfile`: Configuration file for the vagrant based virtual machine setup; already configured for a minimal master-slave setup.
- `cases`: Directory holding all measure cases which can be run in this environment. A `simple` test case is provided as an example.
- `conf`: Directory containing all environment specific configurations. Especially noteworthy is the `chef` subdirectory which holds the full VM configuration scripts. Also contained in this directory are the `xsl` templates used for the measure simplification and report generation and a base `Makefile` to facilitate the creation of measure cases.

- `contrib`: Contributed assets and sources to be used inside the environment. Already contained in this directory is a patched pop-c++ version to avoid raw encoded communications.
- `reports`: The destination of all measure results along with the resulting reports.
- `settings.py`: An environment specific configuration file to allow to override the default settings directives.

Note: For the remaining part of this quick-start we will assume that your current working directory corresponds to the environment root. If this is not yet the case, please `cd <path-to-the-env>`.

2.1.2 Step 2: Setup the virtual machines

Once the testing environment is created, the virtual machines on which the measures will be run have to be created, booted and configured. Fortunately this task is made simple by the `vagrant` tool:

```
$ vagrant up
```

When you execute this command for the first time, the preconfigured virtual machine box will be downloaded from the internet; in this case the following text will be printed during the execution:

```
[master] Provisioning enabled with Vagrant::Provisioners::ChefSolo...
[master] Box lucid32 was not found. Fetching box from specified URL...
[master] Downloading with Vagrant::Downloaders::HTTP...
[master] Copying box to temporary location...
Progress: 1% (5288808 / 502810112)
```

Vagrant is now probably downloading the virtual machine box to your system. This process and the provisioning process described below take a neat amount of time so it could be a good time to take a coffee break.

Once a machine is downloaded and booted, vagrant begins the "provisioning process", which simply means to copy the required chef recipes and resources to the VM and to execute the set of defined actions to configure the host.

Once the complete process is finished, you have two running headless virtual machines ready to execute pop-c services and objects.

Note: You can play with the virtual machines through `ssh`. To connect simply issue:

```
$ vagrant ssh master # s/master/slave/ if you want to connect to the slave instead
```

2.1.3 Step 3: Run the measure

In this quick-start we will run the base example bundled with the newly created environment. Refer to the *Creating measure cases* document to get help on how to create and personalize a new measure case.

The first thing to do when a new measure case is added to the library is to compile it on each virtual machine. To do so, issue the following command:

```
$ pas compile
```

If there is more than one possibility, the `compile` subcommand asks you to choose the measure to compile or, alternatively, you can provide the name of the measure on the command line directly.

When ran, the `compile` subcommand, automatically calls the `build make` target on each known host and makes sure to add the needed informations to a global `obj.map` file.

Once the sources are compiled, we are ready to run our measure. Measuring is done through one or more `tshark` instances per host. `pas` provides commands to start and stop `tshark` based measures on all or on selected hosts/interfaces:

```
$ pas measure start

Only one test case found: simple.
[33.33.33.10] sudo: rm -rf /measures ; mkdir /measures
[33.33.33.10] sudo: screen -dmS simple.lo.lo tshark -i lo -t e -w ...
[33.33.33.10] sudo: screen -dmS simple.eth1.eth1 tshark -i eth1 -t e ...
[33.33.33.11] sudo: rm -rf /measures ; mkdir /measures
[33.33.33.11] sudo: screen -dmS simple.lo.lo tshark -i lo -t e -w ...
```

The `pas measure start` subcommand cleans up the measure destination directory on the target-host and starts a detached named screen session to wrap the `tshark` process. This allows to let measures live between different connections and to terminate them by name.

Now that the measure daemon is running, we can start the `jobmgr` and the actual measure case.

Note: If the initialization done by the `jobmgr` processes is not relevant for the measure, it is of course possible to start the job managers before starting the measure.

To start all job managers on all hosts – and with some automatically provided grace period – issue the following command:

```
$ pas jobmgr start
```

Finally we can also start the previously compiled pop binary and measure the different established connections:

```
$ pas execute
```

Once been through these different steps and having waited for the measured program to terminated, the `jobmgr`'s can be shut down and the measure terminated. In short, this comes back to the following two commands:

```
$ pas jobmgr stop ; pas measure stop
```

Congratulations, you just measured your first pop program using the POP Analysis Suite, but the work is not over yet; all of the assets resulting from the measure process are still dispersed all over your virtual machines. Head up to the next section to learn how to assemble all the files into a readable report.

2.1.4 Step 4: Generate the report

As anticipated above, all of the measures are still scattered over the different virtual machines. The first step which has to be done to generate a report is to collect them in a unique place:

```
$ pas measure collect test_measure
```

This command has the effect to gather all different measure files and place them in an appropriate tree structure inside the `report` directory. The different measures are first grouped by measure case + collection timestamp and then by the IP of the originating virtual machine.

Once all files are collected, we can begin to process them:

```
$ pas report toxml      # Converts all measures to xml documents.

$ pas report simplify   # Simplifes the xml document by stripping
                        # unnecessary informations.

$ pas report decode     # Annotates the xml documents with the decoded
                        # POP Protocol payload.
```

The execution of these commands (the execution order is relevant) produces 3 new files for each `<measure>.raw` file:

- A `<measure>.xml` file, containing the XML representation of the measure as returned by the `tashark` conversion command.

- A `<measure>.simple.xml` file, containing the simplification of the previously converted measure. Only relevant data is preserved.
- A `<measure>.decoded.xml` file, containing the same data of the simple XML version annotated with the decoded POP Protocol payload.

The final step allows to generated an HTML document containing the visual representation of the full measure and some additional information. To launch it run:

```
$ pas report report
```

To display the generated report in your browser, simply open one of the `html` file files found in the `reports/<measure-name>_<timestamp>/report` directory.

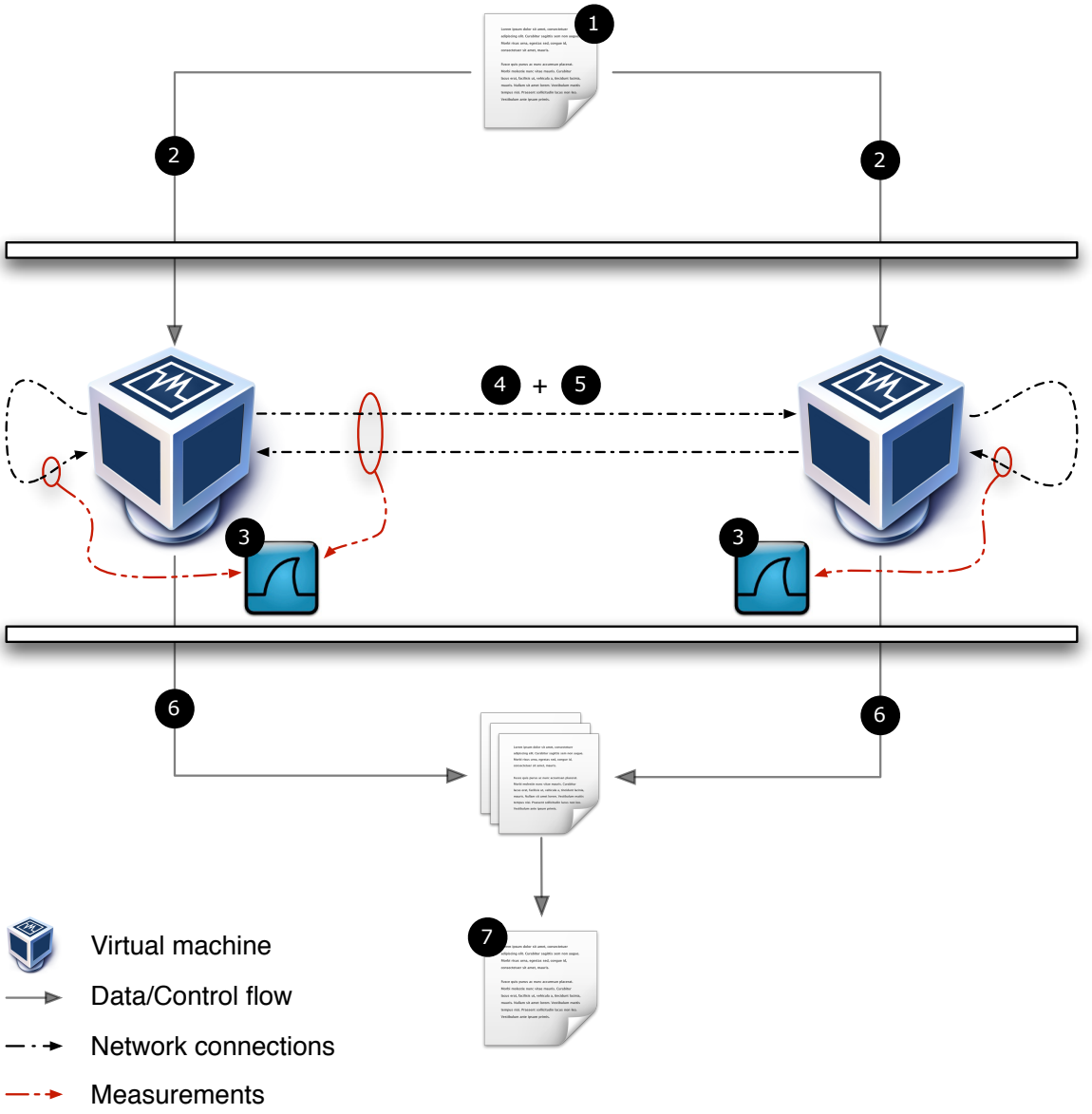
Wow, this was the final step! Sounds like a complicated and tedious process but as you will see by reading the rest of the documentation, much of all this can be automated, allowing to produce a complete report with a single command; hand over to the guide on *composed commands* to find out how.

2.2 Execution model

In the *Quick start* we have seen all steps needed to create an environment, set up the virtual machines, run a measures and generate a report from a practical point of view. In this document we will take a deeper look at the inside workings of the pas tool and its execution model.

Once understood this model and acquired a good overview of the various commands you will be ready to bend the `pas` utility to your will by simply combining or adding different commands.

The following image will give you an high level overview of the execution model of a measure in `pas` and the description of each step contains shortcuts to relevant commands and configuration directive references:



The various steps of the execution model can best be described as follows:

1. **Setup**

   This involves the initialization of the environment, the setup of the virtual machines and the creation of one or more measure cases.

   Related commands: `pas init`, `pas authorize`

   Related settings: `VM_USER`, `VM_PASSWORD`

2. **Distribution and compilation**

   The first part of this step, the *distribution*, is automatically done by sharing the used directories between the host (your machine) and the guests (the virtual machines running on the host).

   The second part, the *compilation* is executed sequentially on each host and produces a binary build of the measure case for each different architecture in your VM setup.

   Related commands: `pas compile`

3. **Beginning of the capturing process**

   A `tshark` instance is started on each configured interface of each guest and every filtered packet captured to a guest-local file.

The `tshark` program is the command-line version of the well known wireshark[1] network protocol analyzer.

Related commands: `pas measure start`, `pas measure stop`, `pas measure kill`

Related settings: `CAPTURE_FILTER`, `INTERFACES`

4. **Job manager startup**

   A `jobmgr` is started on each guest, with a different configuration based on their role. The default setup defines a master and a slave role and the respective configuration to simulate the offloading of objects to a remote guest.

   Related commands: `pas jobmgr start`, `pas jobmgr stop`, `pas jobmgr kill`

   Related settings: `ROLES`, `STARTUP_DELAY`, `SHUTDOWN_DELAY`

5. **POP program execution**

   The developed POP program is executed and the generated traffic captured by the different `tshark` processes.

   Related commands: `pas execute`

   Related settings: `ROLES`

6. **Data collection**

   Once the POP program has executed and returned, the measure and the job managers can be stopped.

   The results of each single measure and the different log files are now spread among all the different involved guests. The collection process retrieves all these files and copies them to a common shared location where the host can access them.

   Related commands: `pas measure collect`

   Related settings: `LOG_FILES`

7. **Report generation**

   Now that the host has all measure results locally available it can process and assemble them into one or more measure reports.

   This process involves multiple steps, such as conversions, simplifications, payload decoding and the final assembly.

   Related commands: `pas report toxml`, `pas report simplify`, `pas report decode`, `pas report report`

   Related settings: `DISPLAY_FILTER`

Once grasped these concepts you are ready to personalize your testing environment. If you have not done so yet, hand over to the *Quick start* for a step-by-step begin-to-end on how guide to take a measure or continue to the detailed guide about how to *create measure cases*.

The *advanced usage* chapter provides documentation on more technical and complicated aspected of the POP Analysis Suite needed to completely customize and adapt an environment to special setups (i.e. a preexisting setup with real machines).

2.3 Creating measure cases

A *measure case* in *PAS* is simply a directory containing a bunch of special files. As we will see in this section, most of this structure is based on a simple naming convention.

The default directory structure looks like the following:

[1] http://wireshark.org

```
+ <environment>/
 \
  + cases/
   \
    + <case-name-1>/
    |\
    | + Makefile
    | + src/
    | |\
    | | + source1.ph
    | | + source1.cc
    | | + main.cc
    | |
    | + types.py
    |
    + <case-name-2>/
    |\
    | + <...>
    |
    + <case-name-n>/
```

The only conventions which have to be respected for the measure case to conform to the base model are the following:

1. Each measure case shall be in its own directory inside the `cases` subdirectory of the environment base directory;
2. Each measure case shall contain a `Makefile` which shall provide some defined targets, see the *Makefile targets* section);
3. Each measure case shall contain a `types.py` file, defining all the types of the exchanged or called objects (see the *Measure case specific types* section).

The fact of placing your sources inside the `src` subdirectory is not obligatory, but as described below, a base `Makefile` is provided and, to use it, some more conventions have to be applied, it is thus better to respect this rule too.

2.3.1 `Makefile` targets

As anticipated above, the measure case specific `Makefile` has to contain some predefined targets to complain with the specification. These targets are explained in greater detail in the next subsections.

To speed up the creation of a new measure case, a base makefile named `Makefile.base` is placed in the environment's `conf` directory. This file can be included by the measure case specific `Makefile` to inherit the already defined targets.

All required targets are already defined by the base makefile. If you are option for the default build process and file organizations and don't need exotic build configuration, including it should suffice to get you started. Your measure case specific makefile will thus looks something like this:

```
include $(ENV_BASE)/conf/Makefile.base
```

Note: The `ENV_BASE` variable contains the absolute path to the environment base directory and is automatically set to the correct value (either for local or remote invocation) by the `pas` command line tool.

The required targets (`build`, `execute`, `clean` and `cleanall`) are further described below, along with their default implementation:

build

Is responsible to build the sources for the architecture on which it is run on, by making sure that multiple builds of different architectures can coexist at the same time. After each build, an `obj.map` file containing the informations relative to *all* builds has also to be created or updated.

The exact location/naming of the produced artifacts is not standardized, as they are needed only by the `execute` target.

The base `Makefile` provides this target by implementing it in the following way:

```
SRCS=src/*.ph src/*.cc
EXEC=$(notdir $(abspath .))
ARCH=$$(uname -m)-$$(uname -p)-$$(uname -s)

build: bin object

dir:
     mkdir -p build/$(ARCH)

bin: dir
     popcc -o $(EXEC) $(SRCS)
     mv $(EXEC) build/$(ARCH)

object: dir
     popcc -object -o $(EXEC).obj $(SRCS)
     mv $(EXEC).obj build/$(ARCH)
     build/$(ARCH)/$(EXEC).obj -listlong >>build/obj.map
     sort -u build/obj.map >build/obj.map.temp
     mv build/obj.map.temp build/obj.map
```

execute

Allows to run an already built POP program on the architecture the target is run on.

This target uses the artifacts produced by a run of the `build` target (thus the binaries and the `obj.map` file) by choosing the correct architecture; it has thus to be able to detect its architecture and to run the respective binary.

The base makefile provides this target by implementing it in the following way:

```
EXEC=$(notdir $(abspath .))
ARCH=$$(uname -m)-$$(uname -p)-$$(uname -s)

execute:
   popcrun build/obj.map build/$(ARCH)/$(EXEC)
```

clean

Cleans up the compiled files for the architecture on which it is run on. As before, this target is also already provided by the base makefile:

```
ARCH=$$(uname -m)-$$(uname -p)-$$(uname -s)

clean:
   rm -rf *.o build/$(ARCH)
```

cleanall

Cleans up all builds for all architectures for its measure case, simply implemented as:

```
cleanall: clean
     rm -rf build
```

2.3.2 How will pas invoke make

As seen in the previous target definitions, the base Makefile makes different assumptions about the CWD at invocation time and about some variables which have to be set in the environment. To completely exploit all features which make makes available, the details of the invocation have to be known.

pas differentiates between local (i.e. on the host machine) and remote (i.e. on a guest or on a remote machine) invocations.

Remote invocations For remote invocations the path is set to the measure case base directory (i.e. a cd to ENV_BASE/cases/<case-name> is done).

Additionally the ENV_BASE environment variable is set to the specific environment base location on the remote host as read from the PATHS settings directive.

Remote invocations are executed by the pas run and pas execute commands.

Local invocations For local invocations the path is left untouched, but both the ENV_BASE and EXEC environment variables are set and the -e flag passed to make in order to let environment variables override local defined variables.

2.3.3 Measure case specific types

It will often happen (if not always) that you create your custom POP-C++ classes for a specific measure case.

The python based parser used to decode the TCP payload cannot know what types are encoded in it without further informations. To fulfill this need, custom types can be created and registered to the parser using the *POP Parsing DSL*.

The *pas report decode* subcommand automatically loads the types.py file contained in the measure case directory and registers the contained types for the decoding session.

The syntax of this file is quite simple and based upon a declarative python syntax; refer to the *POP Parsing DSL* document for further information about it and how to define your custom types.

2.4 Configuration

The pas command line tool can be configured to customize its behavior to better suit the user's needs. To achieve this, it reads configuration settings from two distinct python modules:

1. The first loaded module (which is always loaded) is the pas.conf.basesettings module; it contains all default settings as documented in the *Settings reference*.
2. Additionally, if a local settings module is found (either in the CWD or provided by the *–settings option*), it is loaded too and overrides the values defined by the previous import.

Using this simple mechanism, it is possible to override the default settings on a per-environment basis and to provide your own directives if needed by any custom command.

2.5 Composing commands

As you might have seen in the *Quick start*, running a measure from begin to end is a process involving many different steps. If the environment has already been set up and you are only tweaking your measure case source, launching a whole measurement cycle requires more than 10 different commands. Obviously, running them manually each time rapidly becomes cumbersome.

To solve exactly this problematic, `pas` introduced the concept of *workflow*. A *workflow* is nothing else than a `Makefile` target invoked trough the `pas run` command.

The `pas run` command simply takes care to invoke one or more targets locally with the right environment variables set up and the right working directory, as defined in the *How will pas invoke make* section.

Now that we know how to leverage the power of a `make` inside `pas`, let's take a look at a simple example. We want to be able to run a full measure cycle, from compilation to reporting, with a single command.

This is the command we want to be able to execute:

```
pas run <case-name> measureall
```

Then, the targets to add to the measure case specific `Makefile` to provide the needed interface, would be the following:

```
measureall: compile measure report

compile:
   pas compile $(MEASURE_NAME)

measure:
   pas measure start
   pas jobmgr start
   pas execute $(MEASURE_NAME)
   pas jobmgr stop
   pas measure stop
   pas measure collect $(MEASURE_NAME)

report:
   pas report toxml $(MEASURE_NAME)
   pas report simplify $(MEASURE_NAME)
   pas report decode $(MEASURE_NAME)
   pas report report $(MEASURE_NAME)
```

The `Makefile` of the bundled `simple` measure case already contains three additional targets:

- A `run` target to execute a measure by starting first the measure, then the job managers and in the end running the measure case by tearing it all down and collecting the data when finished.
- A `report` target to generate the HTML report from the raw measures.
- A `reset` target (inherited from the base `Makefile`) which cleans all builds, kills all running measures and job managers and recompiles POP binaries from the sources for all hosts.

This commands composition technique allows us to automate some common workflows and repetitive tasks. For more advanced commands which don't involve only composition, the *custom commands* documentation introduces the creation of commands using the very same tools as the `pas` subsytem.

2.6 Report view

The generated HTML report view presents three different tabs: a *diagram* tab, a *measure logs* tab and a *source code* tab. For the purpose of this guide, we will concentrate ourselves on the main tab, the *diagram* tab.

The diagram tab can be split up in 6 logical regions: tabs, filters, actors, conversations, transactions and details. With the exception of the tab bar, the other 5 regions are described in detail in the following subsections.

2.6.1 Filters

The filter bar allows to trigger different visualization modes of the different transactions, by including or excluding one or more transactions based on some criteria or by rendering a logical group of transactions differently.

The available filters and renderers are:

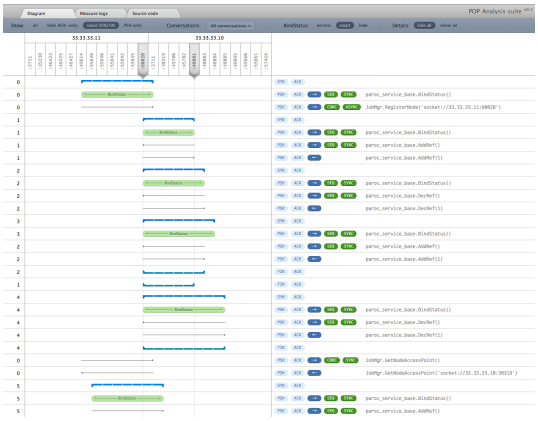


Figure 2.1: Diagram view with the *filters* region highlighted.

Display mode Triggers the visualization of transactions based on their TCP flags. Possible values are *all* to display all transactions, *hide ACK-only* to show all transactions but they with only the `ACK` flag set, *smart SYN/FIN* to group transactions belonging to the same three way handshake (or the same `FIN` sequence) into one and rendering it appropriately or *PSH only* to display only transactions containing actual POP method calls or responses.

Conversations filter A simple filter allowing to select or deselect visible conversations.

BindStatus display mode Triggers the visualization of `BindStatus` calls and responses. Possible values are *normal* to display them as normal transactions, *smart* to group them into a single transaction and render it appropriately or *hide* to completely hide any `BindStatus` related transaction.

Hide/show all details A simple switch to either hide all transaction details or to show them all.

2.6.2 Actors

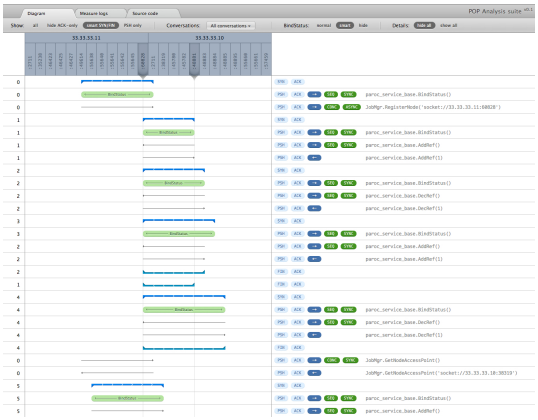


Figure 2.2: Diagram view with the *actors* region highlighted.

The actors region displays the different IP addresses and ports bound to each conversation. When hovering a transaction, the two peers are automatically highlighted.

This highlight mode can be fixed by clicking on a node and reset by clicking a second time.

Additionally it is possible to hide or show all conversations from/to a given actor or by clicking on it by keeping the `alt` or, respectively, the `meta` key respectively pressed.

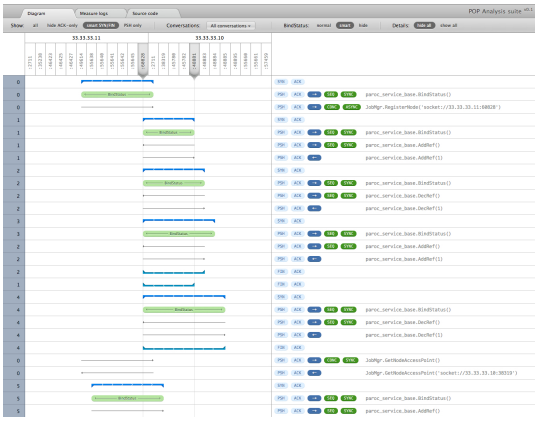


Figure 2.3: Diagram view with the *conversations* region highlighted.

2.6.3 Conversations

The conversations region simply indicates for each transaction to which conversation it belongs to. A conversation is simply the set of all transactions which where transmitted on the same TCP connection.

The conversations are numbered, starting at zero, from the oldest to the youngest and their number corresponds to the number displayed in the *Conversations filter* too.

2.6.4 Transactions

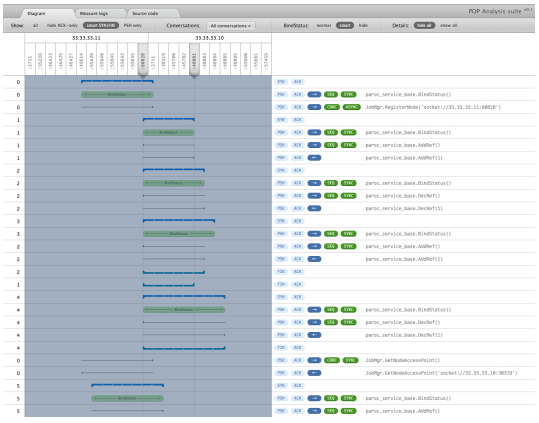


Figure 2.4: Diagram view with the *transactions* region highlighted.

This region displays from which actor to which actor a transaction happened and, if the respective smart display filters are active, renders SYN/FIN and BindStatus requests accordingly.

When overing a transaction, its bounds are automatically highlights in the same way as it was for the actors above.

2.6.5 Details

This region is the most interesting and informational of the entire report.

It provides details such as TCP flags, POP invocation semantics, transaction type, called object and method, passed arguments,... simply click on the corresponding row to expand it and show all details of the transaction.

An example of expanded transaction details is displayed in the figure below; the different parts composing it are explained in deeper detail after the screenshot.

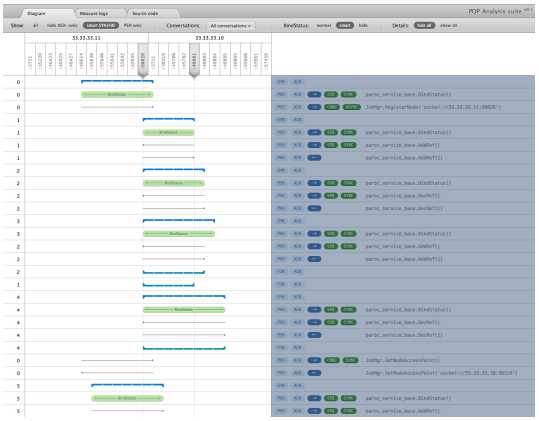


Figure 2.5: Diagram view with the *details* region highlighted.

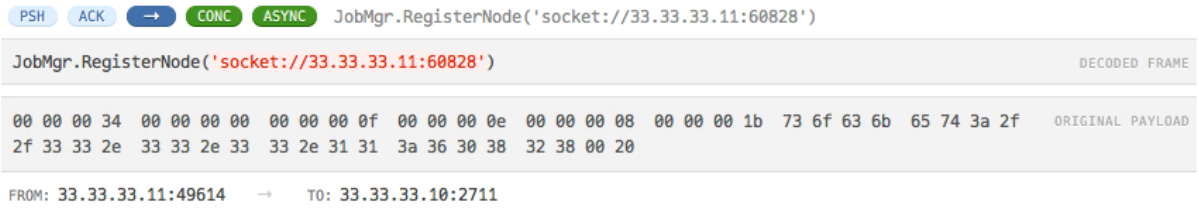


Figure 2.6: Expanded details region for a single transaction.

Starting at the top-left corner, we are presented with the TCP flags which were set for this transaction; possible values are `ACK`, `SYN`, `PSH` and `FIN`.

Just after them, on the right, the transaction type; possible values are → for a method call, <− for a reply and × for an exception.

The next flags group, in green, represents the POP call semantics; possible values for them are the well known `SYNC`, `ASYNC`, `CONC`, `SEQ`, `MTX` (for mutex) and `CONSTRUCT` (for constructor calls).

The rest of the line is dedicated to a short representation of the method call, including class name, method name and the arguments if they are not too long to be represented.

On the following line, inside the *decoded frame* field, the full decoded and highlighted request is represented.

The original payload shows the hex encoded original content of the payload as it was captured by `tshark`.

The last line only remembers, in a textual way, the two involved actors with the respective IP addresses and port numbers.

2.6.6 Other tabs

Other two tabs are available in the tab bar, and although not providing as much information as the diagram view, they can provide some useful insight in the measure constructions, especially for a third party person watching at the report for the first time.

Both views are file browsers but for two different file type groups:

Logs view This view groups all collected log files relative to this specific measure and presents them in a simple to use file browser with no additional syntax highlighting.

Source code view This view groups all source code files found in the original measure case folder and provides syntax highlighting for the different file types such as makefiles, POP-C++ headers, C++ source code,...

CHAPTER

THREE

ADVANCED USAGE

This section is intended for users which have already well understood the basic concepts presented in the *first part* of the documentation and are willing to customize a given testing environment to fulfill special needs.

By personalizing the environment using the directives presented in this chapter, the `pas` tool as to be seen more as a framework for than as a command line tool as the internals are much more exposed and it is possible to interact directly with the basic building blocks of the same library used by the built-in commands.

Four main arguments are presented in this chapter:

1. The creation of subcommands allows to add arbitrary commands to the `pas` utility by loading them from different locations and is described in the first section;
2. The description of the python POP Parsing Domain Specific Language allows to create new complex types and defines classes to perform payload decoding of custom developed POP objects. This argument is treated in the second section;
3. The third section introduces the assumptions the `pas` tool makes about the guest system and indicates therefore how a real machine has to be configured to being able to interact with the `pas` tool;
4. By describing how to add and configure additional virtual machines, the understanding of the fourth section will let you create complex setups with different virtual machines and running measures on them.

3.1 Custom subcommands

The `pas` commands can be grouped into 4 areas of interest:

Meta management Allows to execute all actions to setup and manage a testing environment and its configuration.

Jobmgr management Provides a simple interface to manage remote `jobmgr` processes (start, stop, kill,...).

Measuring Collection of commands to manage the measuring process, such as starting or stopping a measure.

Processing Different tools to act upon al already acquired measure, to transform, simplify, filter, etc. the different assets resulting from a measuring process.

A fifth group, the **derived commands** group can also be added and contains all custom defined workflows resulting from the chaining of different "primitive" commands. This form of custom command creation was already described in the *command composition* section of the first chapter.

3.1.1 Architecture

The various available commands are gathered at runtime by the `pas` binary (oh, well... actually it is a python script) when executed. The entry point recursively scans the `pas.commands` package to retrieve the various commands.

This mechanism allows for great flexibility in the definition of parsers for single commands and makes the addition of new commands relatively easy.

The next section will show you how to create a new command by adding it directly to the `pas` package.

A simple example, the `unreported` subcommand

A `pas` subcommand is built for each module found in the `pas.commands` package. For this operation to succeed, some conventions have to be respected; suppose we want to add a new command to the pas utility (i.e. a command to list all collected measures with no report), then we want to proceed as follows:

1. Create a new python module inside the `pas.commands` package. Name it as you want to name your command. As we want our command to be named `unreported`, we create the `pas/commands/unreported.py` file.

   Note: Filenames starting with underscores are automatically filtered out.

   If you run the `pas` utility with the `unreported` subcommand, then an error is reported indicating that you don't have implemented the command correctly:

   ```
   $ pas unreported
        ERROR: No command found in module pas.commands.unreported
   usage: pas [-h] [--version] [-v] [-q] [--settings SETTINGS_MODULE]
              {authorize,execute,jobmgr,compile,init,measure} ...
   pas: error: invalid choice: 'unreported'
   ```

2. To provide an implementation for a command, simply define a callable named `command` inside your `unreported` module. The callable shall accept one positional argument (which will be set to the command line options) and return a non-true value if the command succeeded:

   ```
   # Inside pas/commands/unreported.py

   def command(options):
      pass
   ```

   If you run the command now, it should exit cleanly without any message, but not much will be done. Let's write a simple implementation:

   ```
   import os
   import glob
   from pas.conf import settings

   def command(options):
       a = '{0}/*'.format(settings.PATHS['shared-measures'][0])
       r = '{0}/*/report'.format(settings.PATHS['shared-measures'][0])

       all_reports = set(glob.glob(a))
       already_reported = set([os.path.dirname(d) for d in glob.glob(r)])

       not_reported = all_reports - already_reported

       if not not_reported:
           print "No unreported measures found ({0} total measures)".format(
                   len(all_reports))
       elif len(not_reported) == 1:
           print "One unreported measure found ({0} total measures)".format(
                   len(all_reports))
       else:
           print "{0} unreported measures found ({1} total measures)".format(
                   len(not_reported), len(all_reports))

       for i, path in enumerate(not_reported):
           print "{0:2d}: {1}".format(i+1, os.path.basename(path))
   ```

3. Suppose we want now to allow the newly created subcommand to accept some optional (or required) flags and arguments; how can we add support for additional command line parsing to the current implementation?

Fortunately the whole `pas` commands subsystem is based around the argparse[1] module and adding options and arguments is straightforward. The `pas` command line parser builder looks if the module containing the command also contains a callable named `getparser` and, if it is the case, it calls it during the parser construction by passing the subcommand specific subparser instance to it.

If we want to add a `--no-list` flag allowing to turn off the list of unreported measures (thus only showing the total counts), we can proceed as follows:

```
import os
import glob
from pas.conf import settings

def getparser(parser):
    parser.add_argument('-n', '--no-list', dest='show_list', default=True,
        action='store_false')

def command(options):
    # [...]

    if options.show_list:
        for i, path in enumerate(not_reported):
            print "{0:2d}: {1}".format(i+1, os.path.basename(path))
```

Refer to the argparse[2] documentation for the syntax and the usage of the module (just remember that the `parser` argument of the `getparser` function is an `argparse.ArgumentParser` instance and the `options` argument passed to the command is an `argparse.Namespace` instance).

The example illustrated above covers the basics of creating a new subcommand for the `pas` command line utility, but some more techniques allows to achieve an higher degree of flexibility, namely *Recursive subcommands* and *External directory scanning*.

3.1.2 Recursive subcommands

As stated in the introduction to the *Architecture* section, the `pas` entry point *recursively* scans the `pas.commands` package for commands. This allows to define `pas` subcommands which have themselves subcommands.

Take as an example the `jobmgr` subcommand. It defines different actions to be taken upon the different instances, such as `start`, `stop` or `kill`. These three actions are defined as subcommands of the `jobmgr` subcommand.

To create a similar grouping structure for your commands collection, it suffices to define your actions as modules inside a package named after the commands collection name.

To reproduce a structure as the one implemented by the `jobmgr` command, the following directory structure may be set up:

```
+ pas/
 \
  + commands/
   \
    + __init__.py
    + command1.py
    + command2.py
    |
    + jobmgr/
    |\
    | + __init__.py
    | + start.py
    | + stop.py
    | + kill.py
```

[1] http://docs.python.org/dev/library/argparse.html
[2] http://docs.python.org/dev/library/argparse.html

```
    |
    + command3.py
```

Then, you can invoke the `jobmgr`'s `start`, `stop` and `kill` subcommands simply by typing this command:

```
$ pas jobmgr start
```

All other conventions (`command` callable, `getparser` callable, argument types,...) presented in the plain command example still hold for nested commands.

3.1.3 External directory scanning

The main weak point of the architecture as it was presented until now is certainly the fact that for the parser to recognize a new command, the command itself has to be placed inside the `pas.commands` package which, depending on the installation, is buried deeply somewhere in the file system and is easily overridden by a reinstallation.

Fortunately, a mechanism has been put in place to allow arbitrary directories to be scanned for commands: the `COMMAND_DIRECTORIES` settings directive contains a list of directories to be scanned for commands *in addition* to the `pas.commands` package.

By adding your custom commands directory to the list you can have them automatically included in the `pas` utility without the need to modify the `pas` source tree directly.

This is useful if you want to add some commands tied to a particular testing environment or – if it is the case – to a particular *measure case*.

Note: You can override built-in commands simply by specifying a directory containing a command with the same name. Note although that **no recursive merge** is done, thus you can't override a single command of a commands collection while retaining all other collection subcommands.

3.1.4 Command utilities

The `pas.commands` package provides some useful utilities to facilitate common tasks when working with command definitions.

The API for these utilities is presented below. For each utility its use case is described and a usage example is provided. To use them in your custom commands definition simply import them from the `pas.commands` package.

Command decorators

The following group of utilities can be used as decorators for the `command` callable:

`pas.commands.`**`nosettings`**(*func*)

Instructs the command dispatcher to not try to load the settings when executing this command.

Normally the command invoker will try to load the settings and will terminate the execution if not able to do so, but some commands don't need the settings module and are specifically crafted to operate outside of an active testing environment.

To instruct the invoker not to try to load the settings, a nosettings attribute with a non-false value can be set on the command function; this decorator does exactly this.

You can use it like this:

```
from pas.commands import nosettings

@nosettings
```

```
def command(options):
    # Do something without the settings
    pass
```

Parameters **func** – The command function for which the settings don't have to be loaded.

pas.commands.**select_case**(*func*)

Adds a `case` attribute to the `Namespace` instance passed to the command, containing a tuple of `(full-path-on-disk-to-the-measure-case, measure-case-basename)`.

The `case` attribute is populated by calling the `pas.case.select()` function.

If a `case` attribute is already set on the `Namespace` its value will be passed to the the `pas.case.select()` function as default value for the selection.

You can use it like this:

```
from pas.commands import select_case

@select_case
def command(options):
    # Do something with the measure case
    print options.case
```

Parameters **func** – The command function for which the a case has to be selected.

pas.commands.**select_measure**(*func*)

Adds a `measure` attribute to the `Namespace` instance passed to the command, containing a tuple of `(full-path-on-disk-to-the-measure, measure-basename)`.

The `measure` attribute is populated by calling the `pas.measure.select()` function.

If a `measure` attribute is already set on the `Namespace` its value will be passed to the the `pas.measure.select()` function as default value for the selection.

You can use it like this:

```
from pas.commands import select_measure

@select_measure
def command(options):
    # Do something with the measure
    print options.measure
```

Parameters **func** – The command function for which the a case has to be selected.

Argument shortcuts

The following group of utilities can be used as factories for common/complex additional arguments/option to bind to a parser instance for a custom subcommand:

pas.commands.**host_option**(*parser*, *argument=False*)

Adds an optional `--host` option to the `parser` argument.

The given hosts can be read using the `hosts` attribute of the `Namespace` instance passed to the command to be executed.

The resulting `hosts` attribute will always be a list of the hosts specified on the command line or an empty list if no hosts were specified.

If the optional `argument` flag is set to true, then this decorators adds an argument insted of an option.

Multiple hosts can be specified using multiple times the `--host` option or by giving multiple `HOST` arguments as appropriate.

Use it like this:

```
from pas.commands import host_option

def getparser(parser):
    host_option(parser)

def command(options):
    # Do something with it
    print options.hosts
```

Parameters

- **parser** – The `ArgumentParser` instance on which the `--host` option shall be attached.
- **argument** – A flag indicating if the hosts shall be parsed as arguments instead.

`pas.commands.`**`case_argument`**(*parser*)

Adds an optional `case` argument to the given parser.

No validations are done on the parsed value as the settings have to be loaded to obtain the local measure-cases path and they can't be loaded before the full command-line parsing process is completed.

To bypass this issue, the `pas.commands.select_case()` command decorator can be used which looks at the `Namespace` instance and does all the necessary steps to provide the command with a valid measure case path.

Use it like this:

```
from pas.commands import case_argument, select_case

def getparser(parser):
    case_argument(parser)

@select_case    # Optional
def command(options):
    # Do something with it
    print options.case
```

Parameters **parser** – The `ArgumentParser` instance on which the `case` argument shall be attached.

`pas.commands.`**`measure_argument`**(*parser*)

Adds an optional `measure` argument to the given parser.

No validations are done on the parsed value as the settings have to be loaded to obtain the local measures path and they can't be loaded before the full command-line parsing process is completed.

To bypass this issue, the `pas.commands.select_measure()` command decorator can be used which looks at the `Namespace` instance and does all the necessary steps to provide the command with a valid measure path.

Use it like this:

```
from pas.commands import measure_argument, select_measure

def getparser(parser):
    measure_argument(parser)

@select_measure    # Optional
def command(options):
```

```
# Do something with it
print options.measure
```

Parameters **parser** – The `ArgumentParser` instance on which the `measure` argument shall be attached.

3.2 POP Parsing DSL

The informations transmitted in the POP encoded messages don't include the type of the transmitted values as these are defined at compilation time and known to both involved parties.

With the use of python and the need to decode message exchanges between arbitrary peers, information about the transmitted values data-types is not available and has to be provided somehow.

The implemented python based parser for POP messages introduces the concept of a *types registry*. Each time a message has to be decoded, the specific types of the payload are retrieved from the registry using the `classid`/`methodid` couple or the exception code contained in the POP payload header.

As the measuring of arbitrary programs introduces new classes, methods and data types, it does not suffice to provide a built-in set of known data types and the need to let the final `pas` user specify custom and complex types arises.

For this purpose a special python-based domain specific language as been defined. The DSL builds upon a few directives to be able to define primitive types (rarely needed), complex types and new classes and to register them to the registry in order to let the parser retrieve them when needed.

Refer to the *Built-in parser types* document for an overview of the already provided types and classes.

3.2.1 Defining classes and methods

Defining a new class and its corresponding methods is an easy task thanks to the declarative syntax offered by the POP Parsing DSL.

Each file which will finish up loaded by the types registry shares a common import:

```
from pas.parser.types import *
```

Once imported the different types, it is possible to use the `pas.parser.types.cls()`, `pas.parser.types.func()` and `pas.parser.types.exc()` functions to create new classes, methods and exceptions respectively.

Their API is the following:

`pas.parser.types.`**`cls`**(*id*, *name*, *methods*)
Allows to define and automatically register a new POP class.

Parameters

- **id** (*int*) – The classid to which this class will be mapped.
- **name** (`str` or `unicode` compatible object) – The name to use as string representation of this class. Normally its the original class name.
- **methods** (list of `pas.parser.types.func()` objects) – The list of methods bound to an instance of this class.

`pas.parser.types.`**`func`**(*id*, *name*, *args*, *retypes*)
Allows to define a new POP method bound to a specific class instance. The method itself will never know to which class it belongs; to bind a method to a class insert it in the `methods` argument at class declaration time.

Parameters

- **id** (*int*) – The method ID to which this method will be mapped.
- **name** (`str` or `unicode` compatible object) – The name to use as string representation of this method. Normally its always the original method name.
- **args** (*list of POP Parser DSL types*) – A list of arguments taken by this method. The provided types will be directly used to decode the payload. Any built-in or custom defined scalar or complex type is accepted.
- **retypes** (*list of POP Parser DSL types*) – A list of types of the values returned by this method. The provided types will be directly used to decode the payload. Any built-in or custom defined scalar or complex type is accepted.

  Note that an `[out]` argument will be present in the POP response and shall thus be inserted into the `retypes` list.

`pas.parser.types.`**`exc`**(*id*, *name*, *properties*)

Allows to define and automatically register a new POP exception type.

Parameters

- **id** (*int*) – The exception ID to which this exception will be mapped.
- **name** (`str` or `unicode` compatible object) – The name to use as string representation of this exception. Normally its always the original exception name.
- **properties** (*list of POP Parser DSL types*) – The list of properties bound to an instance of this exception.

A possible definition of the `paroc_service_base` class using the parser DSL is the following:

```
# Import the POP types and the cls and func helpers
from pas.parser.types import *

# Define the class and its methods
cls(0, 'paroc_service_base', [
    func(0,  'BindStatus',  [],       [int, string, string]),   # broker_receive.cc:183
    func(1,  'AddRef',      [],       [int]),                   # broker_receive.cc:218
    func(2,  'DecRef',      [],       [int]),                   # broker_receive.cc:238
    func(3,  'Encoding',    [string], [bool]),                  # broker_receive.cc:260
    func(4,  'Kill',        [],       []),                      # broker_receive.cc:287
    func(5,  'ObjectAlive', [],       [bool]),                  # broker_receive.cc:302
    func(6,  'ObjectAlive', [],       []),                      # broker_receive.cc:319
    func(14, 'Stop',        [string], [bool]),                  # paroc_service_base.ph:47
])

# Done. No further actions are required
```

By reading the preceding snippet, a few observations can be made:

1. Not all methods are defined. In fact, the parser doesn't care if the object is fully defined or not until it doesn't receive a request or a response for an inexistent method ID.
2. Methods can return multiple values. The first return value is always mapped to the C++ return value (if non-`void`), while the following values are mapped to the arguments which were passed as references.
3. The `bool` type is often used in responses. The actual C++ return type is an `int`, but as it is interpreted only as a success flag a `bool` type has more semantic meaning in these particular cases.

3.2.2 Defining complex and composite types

In the previous section the process of defining a new class and its methods was presented, but the illustrated example was based on a relative simple class.

What has to be done, if for example, the `POPCSearchNode` class has to be defined (more specifically its `callbackResult` method)?

The following is the signature of the `POPCSearchNode::callbackResult` method:

```
conc async void callbackResult(Response resp);
```

As you might have noticed, it takes a `Response` object as argument, so a possible definition using the parser DSL syntax would be:

```
cls(1001, 'POPCSearchNode', [
    # ...other methods here...
    func(38, 'callbackResult', [Response], []),
])
```

But how is the `Response` object defined and how does the parser know how to decode it? The `Response` object is what in the python parser domain is called a *compound type* and fortunately its definition is much easier than you might think; we know that a `Response` object is composed by a `string`, a `NodeInfo` object and an `ExplorationList` object, so its definition becomes:

```
Response = compound('Response',
    ('uid', string),
    ('nodeInfo', NodeInfo),
    ('explorationList', ExplorationList),
)
```

This example introduced two new concepts:

1. The `compound` function, which allows to create composite objects based on a list of (`name`, `type`) tuples;
2. Nested compound objects definition. A compound object can further contain compound objects (the preceding snippet contains `NodeInfo` and `ExplorationList` as subtypes).

If we look deeper in detail, we'll see that the `ExplorationList` object isn't actually a compound object, but rather a list of compound objects.

Fortunately, the task of defining a list is also easy using the shortcuts offered by the DSL:

```
ExplorationList = array(
    compound('ListNode',
        ('nodeId', string),
        ('visited', array(string)),
    )
)
```

As you can see, defining new complex types is an easy task once the syntax and the different *composers* are known. Below you can find the API of the different composers that come built-in with the python POP parser:

`pas.parser.types.`**`compound`**(*name*, **parts*)

Creates a new compound type consisting of different parts. The parts are specified by the `parts` argument and read one adter the other from the POP payload.

Parameters

- **name** – The name to give to the new type, used when pretty printing the value.
- **parts** (list of `(name, type)` tuples) – The actual definition of the compound type.

Use it like this:

```
NewType = compound('NewTypeName',
    ('member_1', string),
    ('member_2', int),
    ('member_3', float)
    # ...
)
```

`pas.parser.types.`**`dict`**(*key*, *value*)

Creates a new complex type mapping keys to values. All keys will share the same value type and so will all values.

Parameters

- **key** – The type to use to decode the key.
- **value** – The type to use to decode the value.

Use it like this:

```
# SomeCompoundType.dict_member will hold a mapping of strings to integers

SomeCompoundType = compound('SomeCompountTypeName',
    # ...other members...
    ('dict_member', dict(string, float)),
)
```

pas.parser.types.**array**(*type*)

Creates a new complex type representing a length prefixed array of homogeneous items.

Parameters **type** – The type of each item in the array.

Use it like this:

```
# SomeCompoundType.array_member will hold a variable length array of ints

SomeCompoundType = compound('SomeCompountTypeName',
    # ...other members...
    ('array_member', array(int)),
)
```

pas.parser.types.**optional**(*type*, *unpack_bool=<function popbool at 0x25a0ab0>*)

Special composer type which allows to declare a structure member as optional. An optional member is member prefixed by a `bool` flag; the flag is first read, if it yields `true`, it means that the value was encoded and it can be read, if it yields `false`, it means that no value was encoded and the member is skipped.

Parameters

- **type** – The type of the optional value.
- **unpack_bool** (*callable*) – The type to use to decode the boolean flag. Defaults to `popbool`, but can be changed to `bool` if the encoding is done in an RPC compliant way.

Use it like this:

```
# SomeCompoundType.optional_member will hold a string if it was encoded
# or None if it was not encoded

SomeCompoundType = compound('SomeCompountTypeName',
    # ...other members...
    ('optional_member', string),
)
```

3.2.3 Defining scalars

In the previous sections we have seen how to create new classes with bound methods and how to define new argument or return types for those by combining scalars into more complex structure in different ways.

The last building block needed to fully grasp the parsing details is the decoding of scalars. It is here that the real work happens, as complex or compound types only arrange scalars in different orders to decode a full structure, but don't tell them *how* to decode the single values.

New scalar types are not needed as much as new complex types, but it can serve to better understand the whole parsing process as well. Furthermore it allows to define how to decode POP-C++ specific types (e.g. the `popbool` primitive) which don't comply with the standards.

A scalar type, as well as all types returned from the different composers seen above, is simply a `callable` which accepts an xdrlib.Unpacker[3] object and returns a decoded python object. Refer to the `Unpacker` documentation for further information about the already supported data types.

As an example, the implementation of the `popbool` type is shown below:

```
import __builtin__

# Define a callable which taks a single argument
def popbool(stream):
    # Read an integer and check that the highest byte is set
    result = stream.unpack_int() & 0xff000000

    # Coearce the value to a boolean (use the __builtin__ pacakge as the
    # previous bool definition overwrote the built-in definition)
    return __builtin__.bool(result)
```

3.3 Running `pas` with real machines

The `pas` tool was originally conceived to be run on a especially configured VM setup, but it is enough flexible to be able to work on any machine if some requirements are satisfied.

This section deals with the different assumptions the `pas` tool makes about the guest machine and described how a real machine has to be set up in order to be compatible with `pas`.

3.3.1 Authentication

All communication between `pas` and the remote host happens through `ssh`. As the VMs are especially created for the measures, they don't have strict security requirements and so password based authentication is used to connect to them.

Furthermore `pas` assumes that all remote machines share the same username and password.

To configure a real machine to be able to authenticate the `pas` tool, two ways are possible:

1. Configure the local host and the remote machine in order to authenticate themselves without user interaction (this is often done by using `publickey` authentication, but other, less secure, methods exist).

   The `ssh` transport layer will always try `publickey` authentication first thus different keys can be exchanged between different hosts and a good security level can be achieved.

2. Create a user for `pas` on each machine giving it the same username and password and set the `VM_USER` and the `VM_PASSWORD` directives accordingly.

Furthermore, `pas` requires `sudo` permissions to execute certain commands. On the default VM configurations, it does not need any password to execute commands as root and the real machines have to be configured to behave the same way.

The commands for which `pas` requires `sudo` privileges are to start, stop and kill the `jobmgr` and `tshark` processes and to copy and delete the files created by these.

3.3.2 Dependencies

When using the default VMs, vagrant automatically sets them up to conform to the `pas` requirements. You can directly use the chef configuration bundled with each environment or manually set up your machines to conform to these requirements.

The following packages are required by `pas`:

- A `popc` installation, configured in accordance with the role assigned to the machine.

[3] http://docs.python.org/library/xdrlib.html#unpacker-objects

- The `tshark` executable, often provided by the system package manager.
- The `screen` executable, often provided by the system package manager, if not already installed.
- A `getip` shell script available on the path which prints the IP address of the interface given as its only argument.

3.3.3 Shared folders

Vagrant allows to easily set up shared folders between the host and one or more guest operating systems. `pas` builds upon this feature to be able to distribute the source code and recover the measure results.

The following folders on the remote hosts have to be shared with the local host:

- `/share/test-cases` on the remote host has to be mapped to the `cases` directory of the test environment.
- `/share/measures` on the remote host has to be mapped to the `reports` directory of the test environment.
- `/share/conf` on the remote host has to be mapped to the `conf` directory of the test environment.

Note: These paths can be configured using the `PATHS` setting directive.

This means that you can adjust these paths to your liking as long as there are three shared folders between the systems and that the settings file is configured accordingly.

Note: If sharing folders is not an option, you can always build custom subcommands to override the commands which need shared folders and do the file transfers over ssh using `scp` or with `rsync`.

3.4 Complex VM setups

The standard configuration provides a master-salve virtual machine setup with two virtual machines. It is clearly possible to add many more machines to a test environment and the task is made very easy by the means provided by the `vagrant` tool.

The process of adding a new virtual machine to the system is a 3-step process:

1. Configure the new virtual machine inside the `Vagrantfile` by following the vagrant documentation[4] or simply by copying one of the existing configuration blocks inside the `Vagrantfile`, like the one presented in the following code snippet:

```
config.vm.define :slaveX do |slaveX_config|
   slaveX_config.vm.network "33.33.33.15"
   slaveX_config.vm.provision :chef_solo do |chef|
      chef.cookbooks_path = "conf/chef/cookbooks"
      chef.roles_path = "conf/chef/roles"
      chef.add_role "slave"
      chef.json.merge!({:popc => {:version => "popc_1.3.1beta_xdr",}})
   end
end
```

   Note: If you opt for the copy and past approach, remember to change the IP address and the name of the new virtual machine.

   ---

2. Boot up and provision the newly created virtual machine. To do so, simply run:

[4] http://vagrantup.com/docs/multivm.html

```
$ vagrant up
```

3. Add the IP of the newly created virtual machine to the `ROLES` settings directive inside the environment specific settings file:

```
ROLES = {
    'master': ['33.33.33.10'],
    'client': ['33.33.33.10'],
    'slaves': ['33.33.33.11', '33.33.33.15'],
}
```

CHAPTER

FOUR

REFERENCES & API DOCUMENTATION

4.1 Commands reference

This section is dedicated to the description of the usage and the effects of the different commands and subcommands bundled with the `pas` command line tool.

4.1.1 The main `pas` command

All operations involving the use of the `pas` command line utility require the use of a subcommand, nonetheless some options common to all subcommands can only be set directly on the main `pas` command.

The usage of the `pas` command (obtained running `pas --help` on the command line) is the following:

```
pas [-h] [--version] [-v] [-q] [--settings SETTINGS_MODULE]
      {authorize,execute,jobmgr,compile,init,measure,report} ...
```

subcommands:

`authorize`
: Exchanges the public keys between all VMs to allow direct ssh connections between them without user input.

`compile`
: Compiles the given test case on all hosts (by default) or on the hosts provided on the command line through the `--host` option.

`execute`
: Executes the given test case on all hosts having a role of `client` (normally only one).

`init`
: Sets up a new measurement and analysis environment in the selected directory (or in the CWD if none is provided).

`jobmgr`
: Commands suite to remotely manage jobmgr instances.

`measure`
: Commands suite to execute measures and collect results.

`report`
: Commands suite process collected measure results.

optional arguments:

-h, --help	Show this help message and exit.
--version	Show program's version number and exit.

-v, --verbose Increments the verbosity (can be used multiple times).

-q, --quiet Decrements the verbosity (can be used multiple times).

--settings SETTINGS_MODULE The path to the settings module

The next two sections are dedicated to the description of some common arguments and options and to the documentation of all built-in subcommands, respectively.

4.1.2 Common arguments and options

`--settings`

The settings option lets you define the path to a directory containing the `settings` module (simply a `settings.py` file). This module is needed by some commands to correctly identify a working environment.

It is possible to use this option when invoking a command from outside the environment working directory.

It is also possible to set the path in the `PAS_SETTINGS_MODULE` shell environment variable instead.

`--verbose` and `--quiet`

These flags increment and decrement the verbosity respectively and can be used multiple times.

The short version `-v` and `-q` flags are also available.

`--host` and `[HOST]`

The host option or its argument alternative syntax are not a general `pas` argument but are used often enough by subcommands to being worth to be documented here.

Many subcommands execute actions on remote machines. By default these actions are carried out on a predefined list of hosts (normally all or all members of a given role). Using this argument or the respective option variant, it is possible to explicitly define one ore more hosts on which executed the command.

Both variants can be used multiple times to provide more than one host.

`MEASURE_CASE`

A similar argument as made for the inclusion of the `host` argument or option in this section can be made for the `MEASURE_CASE` argument too.

The `MEASURE_CASE` argument takes the name of a measure case from the cases directory. In most cases the value is then coerced to a valid value using the `pas.commands.select_case()` command decorator, which means that the value can also be left out and the subcommand will prompt the user to select the desired value from a list of choices.

4.1.3 Built-in subcommands

The following subcommands are already built-in and come bundled with all `pas` installations. In the next section is explained how it is possible to add custom subcommands to an existing `pas` installation, on a global or per-environment basis.

pas init

Usage: `pas init [-h] [DIR]`

positional arguments:

DIR
The directory inside which the environment shall be created.

optional arguments:

-h, --help show this help message and exit

Sets up a new measurement and analysis environment in the selected directory (or in the CWD if none is provided).

The directory has to be empty or inexistent. The command will fail if the directory already exists and is non-empty.

The environment is mainly set up by recursively copying all the files inside pas.conf.suite_template to the newly created directory.

pas authorize

Usage: `pas authorize [-h]`

optional arguments:

-h, --help Show this help message and exit.

Exchanges the public keys between all VMs and adds each host to the `known_hosts` file of each other, in order to allow direct ssh connections between VMs without need for external input.

This command operates only on the default `rsa` key and uses the shared configuration directory to keep a temporary list of all public keys.

The function to which this command is bound doesn't need (and doesn't provide) any specific command-line options and can thus be called in normal library usage without any argument, like in the following example:

```
from pas.commands.authorize import command as authorize

authorize()
```

pas compile

Usage: `pas compile [-h] [--host HOST] [MEASURE_CASE]`

positional arguments:

MEASURE_CASE
Measure case to compile on the different hosts.

optional arguments:

-h, --help Show this help message and exit.

--host HOST Use this option to specify one or more hosts on which this command has to be run. The –host option can be specifed multiple times to define multiple hosts.

Compiles the given test case on all hosts (by default) or on the hosts provided on the command line through the `--host` option.

If no measure case is provided and only one is available, it is compiled; if more are available, the user is presented with a list of available cases and asked to make a choice.

pas execute

Usage: `pas execute [-h] [MEASURE_CASE]`

positional arguments:

MEASURE_CASE
Measure case to execute on all client hosts.

optional arguments:

-h, --help Show this help message and exit.

Executes the given test case on all hosts having a role of `client` (normally only one).

If no measure case is provided and only one is available, it is compiled; if more are available, the user is presented with a list of available cases and asked to make a choice.

pas jobmgr start

Usage: `pas jobmgr start [-h] [HOST [HOST ...]]`

positional arguments:

HOST
Use this option to specify one or more hosts on which this command has to be run.

optional arguments:

-h, --help Show this help message and exit.

Starts the `jobmgr` on one or more hosts.

If no hosts are provided, then the command first starts the `master` hosts and then, after accounting for a startup delay, the `child` hosts.

If one or more hosts are provided, the `start` command is sent to all hosts at the same time.

pas jobmgr stop

Usage: `pas jobmgr stop [-h] [HOST [HOST ...]]`

positional arguments:

HOST
Use this option to specify one or more hosts on which this command has to be run.

optional arguments:

-h, --help Show this help message and exit.

Stops the `jobmgr` on one or more hosts.

If no hosts are provided, then the command first stops the `child` hosts and then, after accounting for a shutdown delay, the `master` hosts.

If one or more hosts are provided, the `stop` command is sent to all hosts at the same time.

pas jobmgr kill

Usage: `pas jobmgr kill [-h] [HOST [HOST ...]]`

positional arguments:

HOST
Use this option to specify one or more hosts on which this command has to be run.

optional arguments:

-h, --help Show this help message and exit.

Kills the `jobmgr` on one or more hosts. If no hosts are provided, the job managers are killed on all known hosts.

pas measure start

Usage: `pas jobmgr start [-h]`

optional arguments:

-h, --help Show this help message and exit.

Starts a new measure on all hosts and for each interface as defined in the `INTERFACES` and `ROLES` settings directives.

The measure is started in a detached named screen session to be able to disconnect from the remote host while letting the measure run. The same name can then also be used to cleanly stop or to kill the measure.

All existing measure files in the remote measure desintation directory will be erased without asking for confirmations! Make sure to have collected all previously run measures you don't want to lose.

Note that it is not possible to run parallel measures (and it doesn't make much sens, as both tshark process will capture all the traffic).

pas measure stop

Usage: `pas jobmgr stop [-h]`

optional arguments:

-h, --help Show this help message and exit.

Stops the measures running on all hosts and for each interface as defined in the `INTERFACES` and `ROLES` settings directives.

The measures are stopped by sending the `quit` command to each named screen session which matches the MEASURE_NAME and interface couple.

pas measure kill

Usage: `pas measure kill [-h] [HOST [HOST ...]]`

positional arguments:

HOST
Use this option to specify one or more hosts on which this command has to be run.

optional arguments:

-h, --help Show this help message and exit.

Kills all the running measures on the given host (or on all hosts if no host is given) independently from the capturing interface or the name with which they were started.

pas measure collect

Usage: `pas measure collect [-h] MEASURE_NAME`

positional arguments:

MEASURE_NAME
The name to give to the collected measure.

optional arguments:

-h, --help Show this help message and exit.

Collects the results of a measure by copying the remote files resulting from the last run measure to the local measures folder and by organizing them by host ip address.

In addition to the measures, the log files specified by the `LOG_FILES` settings directive are also copied.

This process does not alter the remote files, it is thus possible to repeat a `collect` command different times until a new measure is not started.

pas report toxml

Usage: `pas report toxml [-h] [MEASURE]`

positional arguments:

MEASURE
The name of the collected measure to convert to xml.

optional arguments:

-h, --help Show this help message and exit.

Converts the results of an already collected measure from the libpcap format to the corresponding XML format using the `tshark` command line utility.

If `MEASURE` is not provided, the user is presented with a list of already collected measures and asked to make a choice.

pas report simplify

Usage: `pas report simplify [-h] [MEASURE]`

positional arguments:

MEASURE
The name of the collected measure to simplify.

optional arguments:

-h, --help Show this help message and exit.

Converts the results of an already collected and converted measure from the wireshark XML format to a simplified POP oriented XML format.

If `MEASURE` is not provided, the user is presented with a list of already collected measures and asked to make a choice.

pas report decode

Usage: `pas report decode [-h] [MEASURE] [MEASURE_CASE]`

positional arguments:

MEASURE
The name of the collected measure to decode.

MEASURE_CASE
The name of the measure case from which this measure was run. This is needed to be able to use the additionally defined types for the measured binary.

optional arguments:

-h, --help Show this help message and exit.

Decodes the results of an already collected and simplified measure by annotating the XML structure with the needed POP metadata, as extracted from the TCP payload.

The decoding of additional custom types can be defined on a per-measure-case basis and is bast described in the *parser DSL* document.

If `MEASURE` is not provided, the user is presented with a list of already collected measures and asked to make a choice.

`pas report report`

Usage: `pas report report [-h] [MEASURE]`

positional arguments:

`MEASURE`
The name of the collected measure for which to generate a report.

optional arguments:

-h, --help Show this help message and exit.

Generates an HTML report for each already collected, converted, simplified and decoded measure files of the measure named `MEASURE`.

`pas run`

Usage: `pas run [-h] [MEASURE_CASE] [TARGET [TARGET ...]]`

positional arguments:

`MEASURE_CASE`
The name of the measure case of which run the `Makefile`.

`TARGET`
The target to execute, defaults to the default `Makefile` target.

optional arguments:

-h, --help Show this help message and exit.

Locally runs the given targets of the `Makefile` of the given measure case.

This command is useful to create custom commands by simply combining multiple togheter. Refer to the *composing commands* section for more informations about this argument.

4.1.4 Custom subcommands

The `pas` utility allows to define custom commands to extend the built-in set of commands. The process of creating a new command and adding it to the `pas` utility is described in detail in the *Custom subcommands* document.

4.2 Settings reference

As seen in the *Configuration* document, `pas` provides a global `Settings` object which can be populated by loading directives from different sources. `pas` provides a set of default configuration directives (explained in detail below). All of these settings can be modified (and new directives added) by overriding them in the environment-specific `settings.py` file.

The `pas` command line tool automatically loads these directives at startup and then, once read the `--settings` command line options, will load the environment-specific settings file letting it override the already defined values or adding new ones (for example for your *Custom subcommands*).

The syntax of the settings file is simple: each uppercase public (not starting with an underscore) member is set as an attribute of the global `Settings` object.

`pas.conf.basesettings.`**`CAPTURE_FILTER`** **= "'tcp and not tcp port 22"'**
Capture filter to use with the `tshark` command line tool when running a new measure. Note that this is different from the display filter directive (and also uses a different syntax).

The capture filters[1] page on the wireshark wiki provides more informations about this specific syntax.

The default filter captures all TCP traffic excluding `ssh` connections on the default port (22).

`pas.conf.basesettings.`**`COMMAND_DIRECTORIES`** **= ('commands',)**
List of external directories to scan for additional commands. Refer to the *Custom subcommands* document for more information about the creation and use of custom commands.

`pas.conf.basesettings.`**`DISPLAY_FILTER`** **= "'tcp.flags.syn == 1 || tcp.flags.push == 1 || tcp.flags.fin == 1 || tcp.flags.**
Display filters to use with the `tshark` command line tool when converting the measures to xml. Note that this is different from the capture filter directive (and also uses a different syntax).

The display filters[2] page on the wireshark wiki provides more informations about this specific syntax.

The default filter actually does not filter anything (all non tcp traffic is already filtered out by the capture filter) but provides a good starting point for advanced tcp filtering.

Note: It is considered good practice to move as much filtering as possible to the capture phase. Sadly the capture filters don't offer the powerful filtering expressions which are indeed possible using the display filters syntax.

`pas.conf.basesettings.`**`INTERFACES`** **= {'client': [], 'master': ['eth1', 'lo'], 'slaves': ['lo']}**
A mapping of role names to interfaces. A measure will be started for each interface of each machine of a given role. The `pas.conf.map_interfaces()` function provides an easy way to create a list of (`hostname`, `interfaces`) tuples.

`pas.conf.basesettings.`**`LOG_FILES`** **= ('/tmp/jobmgr_stdout_*', '/tmp/jobmgr_stderr_*', '/tmp/paroc_service_log',**
List of path to log files on the remote host which have to be copied along with the measures during the collect phase.

The paths are copied using a simple `cp` command, it is thus possible to use shell variables and substitutions pattern therein.

Note: The `pas.commands.jobmgr.start` command deletes these files each time it is executed.

`pas.conf.basesettings.`**`PATHS`** **= {'local-measures': [None, '/measures'], 'configuration': ['conf', '/share/conf'], 'sha**
General path configuration. Contains a mapping between symbolic path name (i.e. `test-cases`) to a tuple of its local (i.e. `cases`) and remote (i.e. `/share/test-cases`) equivalent.

With the exception of the `local-meaures` entry, these path are all shared using the `Vagrantfile` `config.vm.share_folder` directive.

Note: This settings directive is available mainly to centralize the path configuration and is not meant to be modified. Override these values only if you are sure of what you are doing.

`pas.conf.basesettings.`**`ROLES`** **= {'client': [], 'master': [], 'slaves': []}**
A mapping of role names to IP addresses. The roles are defined as follows:

master Guest machines running a jobmgr instance configured as a master node. The job managers on these machines are started before and stopped after the job managers on slave machines.

Normally these machines run a job manager configured with 0 available object slots, in order to delegate parallel objects execution to slaves.

[1] http://wiki.wireshark.org/CaptureFilters
[2] http://wiki.wireshark.org/DisplayFilters

slaves Guest machines running a jobmgr instance configured as a slave node. The job managers on these machines are started after and stopped before the job managers on master machines.

client Machines on which the measure case if first started.

Note: An IP address can appear more than one time in different roles. For example, a master node can also be a client (this is the default for newly created environments).

`pas.conf.basesettings.`**SHUTDOWN_DELAY = 2**
Delay to introduce between slave `jobmgr` shutdown and master `jobmgr` shutdown, in seconds.

`pas.conf.basesettings.`**STARTUP_DELAY = 2**
Delay to introduce between master `jobmgr` startup and slave `jobmgr` startup, in seconds.

`pas.conf.basesettings.`**VM_PASSWORD = 'vagrant'**
Password to use to connect to a remote machine through `ssh`, is only used if `publickey` authentication is not available or if it fails.

`pas.conf.basesettings.`**VM_USER = 'vagrant'**
Username to use to connect to a remote machine through `ssh`.

4.3 Built-in parser types

The base parser implementation comes with a rich collection of both scalar and composite built-in types and classes to decode standard POP messages not involving newly defined classes.

4.3.1 Scalars

The set of provided scalars include:

string
A simple `parocstring` XDR compatible decoder.

uint

int

float

popbool
POP specific boolean decoder.

The POP-C++ implementation doesn't encode booleans as defined by the XDR RFC. This alternate implementation provides a workaround.

bool
The XDR compliant equivalent of the `popbool` primitive.

4.3.2 Compound types

The set of provided complex and compound types include:

accesspoint

string **endpoint**

NodeInfo

string **nodeId**

string **operatingSystem**

float **power**

int **cpuSpeed**

float **memorySize**

int **networkBandwidth**

int **diskSpace**

string **protocol**

string **encoding**

ExplorationListNode[] **ExplorationList**

ExplorationListNode

string **nodeId**

array(string) visited

ObjectDescription

float **power0**

float **power1**

float **memory0**

float **memory1**

float **bandwidth0**

float **bandwidth1**

float **walltime**

int **manual**

string **cwd**

int **search0**

int **search1**

int **search2**

string **url**

string **user**

string **core**

string **arch**

string **batch**

string **joburl**

string **executable**

string **platforms**

string **protocol**

string **encoding**

dict(string, string) attributes

Request

string **uid**

int **maxHops**

optional(string) nodeId

optional(string) operatingSystem

optional(int) minCpuSpeed

optional(int) hasExpectedCpuSpeedSet

optional(float) minMemorySize

optional(float) expectedMemorySize

optional(int) minNetworkBandwidth

optional(int) expectedNetworkBandwidth

optional(int) minDiskSpace

optional(int) expectedDiskSpace

optional(float) minPower

optional(float) expectedPower

optional(string) protocol

optional(string) encoding

ExplorationList **explorationList**

Response

string **uid**

NodeInfo **nodeInfo**

ExplorationList **explorationList**

POPCSearchNode

ObjectDescription **od**

accesspoint **accesspoint**

int **refcount**

POPCSearchNodeInfo

string **nodeId**

string **operatingSystem**

float **power**

int **cpuSpeed**

float **memorySize**

int **networkBandwidth**

int **diskSpace**

string **protocol**

string **encoding**

4.3.3 Classes

The set of provided classes include:

class paroc_service_base

BindStatus ( ) → int, string, string

AddRef ( ) → int

DecRef ( ) → int

Encoding (*string*) → bool

Kill ( ) → void

ObjectAlive ( ) → bool

ObjectAlive ( ) → void

Stop (*string*) → bool

class CodeMgr

RegisterCode (*string*, *string*, *string*) → void

QueryCode (*string*, *string*) → string, int

GetPlatform (*string*) → string, int

class RemoteLog

Log (*string*) → void

class ObjectMonitor

ManageObject (*string*) → void

UnManageObject (*string*) → void

CheckObjects ( ) → int

class JobCoreService

CreateObject (*string*, *string*, *ObjectDescription*, *int*) → int, string, int

class JobMgr

JobMgr (*bool*, *string*, *string*, *string*, *string*) → void

RegisterNode (*string*) → void

Reserve (*ObjectDescription*, *int*) → float, int

ExecObj (*string*, *ObjectDescription*, *int*, *int*, *int*, *string*) → int, string, int

GetNodeAccessPoint ( ) → string

class AppCoreService

AppCoreService (*string*, *bool*, *string*) → void

class POPCSearchNode

POPCSearchNode (*string*, *bool*) → void

setJobMgrAccessPoint (*string*) → void

getJobMgrAccessPoint () → string

setPOPCSearchNodeId (*string*) → void

getPOPCSearchNodeId () → string

setOperatingSystem (*string*) → void

getOperatingSystem () → string

setPower (*float*) → void

getPower () → float

setMemorySize (*float*) → void

getMemorySize () → int

setNetworkBandwidth (*int*) → void

getNetworkBandwidth () → int

addNeighbor (*POPCSearchNode*) → POPCSearchNode

launchDiscovery (*Request*, *int*) → array of POPCSearchNodeInfo

askResourcesDiscovery (*Request*, *string*, *string*) → void

callbackResult (*Response*) → void

class ParentProcess

callback () → int, string

Note: For the provided classes not all methods are defined yet. The provided definitions suffices for most basic measures involving other external types, but it can be that some requests for more specific measures could not be decoded without extending the method definitions.

4.4 Internal API

4.4.1 pas.case

Collection of utilities to deal with the management of the different test cases of the suite.

pas.case.**compile**(*name*, *localdir=None*, *remotedir=None*)
Compiles the given test case on all the hosts in the system.

The building details and the respect of the convention are enforced by the Makefile and not further explained here.

pas.case.**execute**(*name*, *remotedir=None*)
Executes the given test case on all client hosts (normally onle one).

pas.case.**select**(*name=None*, *basedir=None*)
Scans the basedir (or the test-cases directory defined in the settings) for directories and returns a choice based on different criteria:

1.No directories are found; raise a RuntimeError

2.The name is set; check if it was found and if so return it

3.Only one directory is found; return the found directory

4.Multiple directories are found; ask the user to pick one

1. If no measure case exist yet, a `RuntimeError` is thrown;
2. If only one measure case is found, its path is returned;
3. If more than one measure cases are found, then the user is asked to choose one between them.

The list presented to the user when asked to choose a case will be something like the following:

```
Multiple test cases found:

     [0]: simple
     [1]: complex

Select a test case: _
```

4.4.2 pas.conf

Settings support for the whole pas package.

This module contains methods to load and temporarily store setting directives for the current execution.

Various shortcuts to access the different settings in a usage-oriented way are also provided.

`pas.conf.`**`all_hosts`**`()`
Returns a set of all hosts obtained by chaining all hosts in the ROLES settings directive.

`pas.conf.`**`role`**`(`*role_name*`)`
Returns a list of all hosts for a given role.

`pas.conf.`**`map_interfaces`**`()`
Returns a dict mapping host connection strings to interfaces, obtained by combining the INTERFACES and ROLES settings directives.

4.4.3 pas.env

General test suite environment management functions.

`pas.env.`**`setup`**`(`*dstdir*`)`
Sets up a new environment in the given directory by recursively copying all the files inside pas.conf.suite_template to the given directory.

The destination directory must already exist.

4.4.4 pas.jobmgr

Remote JobMgr management utilities.

The following functions provide some wrappers around remote shell commands to easily start, stop, restart or otherwise interact with a POP-C++ job manager.

`pas.jobmgr.`**`kill`**`()`
Kills ALL running job managers (and the relative search nodes) on the hosts provided by the context or (by default) on all known hosts.

`pas.jobmgr.`**`killall`**`()`
Alias for the kill function, as no special treatment is needed here.

`pas.jobmgr.`**`restart`**`()`
Stops and restarts a currently running JobMgr instance on one or more (depending on the current context) remote nodes in a unique command.

This function is intended to be used to restart a single node. To restart all nodes of a system, use the startall function, which introduces some delays to allow a correct registration of the slaves by the master.

pas.jobmgr.**restartall**()
Restarts all nodes in the system using the stopall and startall functions.

Due to the introduction of the delays, the stop and start calls will not happen in the same command as for the restart function.

pas.jobmgr.**start**()
Starts a JobMgr instance on one or more (depending on the current context) remote nodes.

This function is intended to be used to start a single node. To start all nodes of a system, use the startall function, which introduces some delays to allow a correct registration of the slaves by the master.

pas.jobmgr.**startall**()
Starts all nodes of the system grouped by roles with the necessary delays to allow a proper registration to the parent JobMgr.

The delay between the invocations can be set in the settings.

pas.jobmgr.**stop**()
Stops a currently running JobMgr instance on one or more (depending on the current context) remote nodes.

This function is intended to be used to stop a single node. To stop all nodes of a system, use the startall function, which introduces some delays to allow a correct registration of the slaves by the master.

pas.jobmgr.**stopall**()
Stops all nodes of the system grouped by roles with the necessary delays to allow a proper registration to the parent JobMgr.

The delay between the invocations can be set in the settings.

Note that in this case the delays are not as important as in the start function on could probably safely be omitted. The behavior is preserved to grant compatibility with future versions of the JobMgr which possibly wants to execute some cleanup code before terminating.

In the meanwhile it is possible to set the delay to 0 in the settings.

4.4.5 pas.measure

Measure management functions.

pas.measure.**collect**(*name*, *overwrite=False*)
Moves the relevant files to the shared directory by asking to empty the destination directory if needed.

pas.measure.**decode**(*name*, *measure_case*, *prettyprint=False*)
Decodes the simplified XML representation of the given measure by adding a "decoded" element to each packet containing a payload.

The decoding is done using an XSL transformation coupled with an xslt python extension function which provides the "decoded" element given a payload text string.

pas.measure.**kill**()
Alias for tshark.kill

pas.measure.**report**(*name*, *measure_case*)
Assembles all the acquired resources (such as source code, measures and log files) and generates an html page suitable for human interaction and analysis.

pas.measure.**select**(*name=None*, *basedir=None*)
Scans the basedir (or the shared-measures directory defined in the settings) for directories and returns a choice based on different criteria:

1.No directories are found; raise a RuntimeError

2.The name is set; check if it was found and if so return it

3.Only one directory is found; return the found directory

4.Multiple directories are found; ask the user to pick one

pas.measure.**simplify**(*name*, *prettyprint=True*)
Simplifies all the measure files in pdxml format of the given measure, converting them using the simplify XSL stylesheet. Old simplifications will be overwritten.

If the prettyprint optional argument is True, the result will be formatted using the xmllint tool.

pas.measure.**start**(*name*)
Start a new named measure session in background on all interested hosts.

The hosts are retrieved from the ROLES setting directive and a measure is started for each one.

pas.measure.**stop**(*name*)
Start a new named measure session in background on all interested hosts.

As for the start function, the hosts are retrieved from the interfaces setting directive and the stop command issued on each one.

pas.measure.**toxml**(*name*)
Converts all raw measure files for the given measure to xml using a remote tshark command.

This will overwrite all already converted files with matching names.

4.4.6 pas.shell

Local and remote shell commands execution.

These functions are mainly wrappers around those found in the fabric library.

pas.shell.**ignore_warnings**()
Returns a fabric context manager configured to silently ignore all warnings about running commands.

pas.shell.**workon**(*hosts*)
Returns a fabric context manager which sets the hosts list to the given list.

If the passed hosts argument is a single string, it is silently converted to a list before.

pas.shell.**local**(*cmd*)
Executes a local command without capturing and returning the output.

Thin wraper around the fabric.operations.local function

pas.shell.**remote**(*cmd*, *pty=False*, *user=None*, *sudo=False*)
Executes the given remote command on the hosts list set in the current context.

If user is given or sudo evaluates to True, then a fabric.operations.sudo call is made, otherwise a simple fabric.operations.run.

pas.shell.**cd**(*path*)
Context manager that keeps directory state when calling *run*/*sudo*.

Any calls to *run* or *sudo* within the wrapped block will implicitly have a string similar to `"cd <path> && "` prefixed in order to give the sense that there is actually statefulness involved.

Because use of *cd* affects all *run* and *sudo* invocations, any code making use of *run* and/or *sudo*, such as much of the `contrib` section, will also be affected by use of *cd*. However, at this time, *get* and *put* do not honor *cd*; we expect this to be fixed in future releases.

Like the actual 'cd' shell builtin, *cd* may be called with relative paths (keep in mind that your default starting directory is your remote user's `$HOME`) and may be nested as well.

Below is a "normal" attempt at using the shell 'cd', which doesn't work due to how shell-less SSH connections are implemented – state is **not** kept between invocations of *run* or *sudo*:

```
run('cd /var/www')
run('ls')
```

The above snippet will list the contents of the remote user's `$HOME` instead of `/var/www`. With *cd*, however, it will work as expected:

```
with cd('/var/www'):
    run('ls') # Turns into "cd /var/www && ls"
```

Finally, a demonstration (see inline comments) of nesting:

```
with cd('/var/www'):
    run('ls') # cd /var/www && ls
    with cd('website1'):
        run('ls') # cd /var/www/website1 && ls
```

Note: This context manager is currently implemented by appending to (and, as always, restoring afterwards) the current value of an environment variable, `env.cwd`. However, this implementation may change in the future, so we do not recommend manually altering `env.cwd` – only the *behavior* of *cd* will have any guarantee of backwards compatibility.

`pas.shell.`**`prompt`**(*text*, *key=None*, *default=''*, *validate=None*)

Prompt user with `text` and return the input (like `raw_input`).

A single space character will be appended for convenience, but nothing else. Thus, you may want to end your prompt text with a question mark or a colon, e.g. `prompt("What hostname?")`.

If `key` is given, the user's input will be stored as `env.<key>` in addition to being returned by *prompt*. If the key already existed in `env`, its value will be overwritten and a warning printed to the user.

If `default` is given, it is displayed in square brackets and used if the user enters nothing (i.e. presses Enter without entering any text). `default` defaults to the empty string. If non-empty, a space will be appended, so that a call such as `prompt("What hostname?", default="foo")` would result in a prompt of `What hostname? [foo]` (with a trailing space after the `[foo]`.)

The optional keyword argument `validate` may be a callable or a string:

- If a callable, it is called with the user's input, and should return the value to be stored on success. On failure, it should raise an exception with an exception message, which will be printed to the user.
- If a string, the value passed to `validate` is used as a regular expression. It is thus recommended to use raw strings in this case. Note that the regular expression, if it is not fully matching (bounded by `^` and `$`) it will be made so. In other words, the input must fully match the regex.

Either way, *prompt* will re-prompt until validation passes (or the user hits `Ctrl-C`).

Examples:

```
# Simplest form:
environment = prompt('Please specify target environment: ')

# With default, and storing as env.dish:
prompt('Specify favorite dish: ', 'dish', default='spam & eggs')

# With validation, i.e. requiring integer input:
prompt('Please specify process nice level: ', key='nice', validate=int)

# With validation against a regular expression:
release = prompt('Please supply a release name',
        validate=r'^\w+-\d+(\.\d+)?$')
```

`pas.shell.`**`confirm`**(*question*, *default=True*)

Ask user a yes/no question and return their response as True or False.

`question` should be a simple, grammatically complete question such as "Do you wish to continue?", and will have a string similar to " [Y/n] " appended automatically. This function will *not* append a question mark for you.

By default, when the user presses Enter without typing anything, "yes" is assumed. This can be changed by specifying `default=False`.

4.4.7 pas.tshark

Various utilities to deal with remote tshark operations with support for features such as background execution, stop on request, and advanced tshark interactions.

Note that all of the commands which operate on remote hosts, respect the current set context to retrieve the host lists. When this list is not set or is empty, the command will be run on all hosts. Refer to the documentation on pas.shell to obtain further information.

Most of the command below use the tshark command-line tool directly; refer to its man page for the details about the usage.

`pas.tshark.`**`kill`**`()`
Kills ALL running measures on the hosts provided by the context or (by default) on all known hosts.

`pas.tshark.`**`pcaptoxml`**`(`*infile*, *outfile*, *display_filter=''*`)`
Converts the pcap input file to XML and writes the output to outfile while filtering using the given display filter.

Refer directly to the tshark man page for further informations about the display filter syntax.

`pas.tshark.`**`start`**`(`*name*, *iface='lo'*, *outfile='-'*, *capture_filter=''*`)`
Starts a new tshark capture in background using a named screen session.

The name of the spawned screen session will be the provided name joined with the interface by a dot.

`pas.tshark.`**`stop`**`(`*name*, *interface*`)`
Stops the named tshark capture session on the given interface.

The final name passed to the screen command will be the name joined with the interface by a dot.

4.4.8 pas.xml

class `pas.xml.`**`Transformation`**`(`*stylesheet*`)`
Object oriented wrapper for XSL transformations using lxml.

`pas.xml.`**`prettyprint`**`(`*infile*, *outfile=None*, *local=True*`)`
Reformats an xml document using xmllint.

CHAPTER

FIVE

GLOSSARY

complex type, complex types, composite type, composite type, compound type, compound types Special data types for the POP Parsing DSL which are made up of different scalar or complex types.

composer, composers A function which allows to create complex or composite types using the *POP Parsing DSL* syntax.

CWD The abbreviation for *current working directory*.

DSL The abbreviation for *Domain Specific Language*.

measure, measures The results of a measure case which has been run.

measure case, measure cases A template for a measure. Contains source code, building instructions, measure workflow and optional commands to run and report the results of the measure case.

test case may be used as a synonym.

PAS, pas The abbreviation of *POP Analysis Suite*.

POP, POP model The abbreviation of *Parallel Object Programming* model.

role The role of a remote machine as seen by `pas`. The default configuration defines and leverages the usage of a `client` role to run the POP application, a `master` role where the master `jobmgr` instance is run on and a `slave` role on which the slave `jobmgr` instances are run on.

test case, test cases A synonym for *measure case*

testing environment A directory structure as set up by running the `pas init` command containing all the needed assets to run *measure cases* and report results.

A testing environment normally contains a set of custom virtual machines, some related *measure cases*, environment specific settings and the results of the different *measures*.

virtual machine box A virtual machine box is a package containing a disk image with a preconfigured operating system on it. It can either already be present on the host system or automatically downloads from the internet when needed.

VM, VMs The abbreviation for *virtual machine* and its plural form, respectively.

Todo

Add more terms to the glossary

INDEX

L

M

N

O

P

Q

R

S

T

U

V

W