Viri

Remote execution of Python scripts

Final Project

Marc Albert Garcia Gonzalo / Francesc Guim Bernat
Departament d'arquitectura de computadors
Enginyeria Tècnica en Informàtica de Gestió
Universitat Oberta de Catalunya

June 15th, 2011

Table of Contents

Introduction

This document describes the idea for a new software application, to be developed as the final project of the studies of Computer Engineering at the Universitat Oberta de Catalunya [1], for the department of Computer Architectures.

The document contains relevant information on the motivation for the project, achieved objectives, as well as a detailed planning of execution containing time and cost estimations.

The idea behind this project, is to build an application able to distribute and execute customizable tasks, from one host to others. This is specially useful when managing a large set of host, like a cluster. With this aim, a software daemon, a command line utility and a web user interface will be developed. This whole application will let a system administrator to implement scripts, and to distribute and execute them in a set of hosts in a very efficient way. This can save time of repetitive work, if uploading and executing the script to lots of computers has to be done manually. Also, running the scripts can be automated, becoming a planned and organized task.

Motivation

There are two main motivations for this project. First one is an academic motivation for learning and understanding an application like this one, consisting of several components, interacting and working together, and also, to learn the necessary technologies for implementing it. Second is to provide a useful and powerful tool, that can save lots of time to system administrators of clusters or other kinds of computer infrastructures.

Background and similar applications

While there is still a strong habit to use a single computer to run a single application, this model is not optimal, and it is being replaced by another where applications can scale, and grow as the user base grows. This new model allows optimizing hardware and electricity resources, by maximizing the resources of our computer infrastructure.

While this new approach is getting popular, it is more important to work efficiently with large sets of machines.

Several applications can be found for remote executing single commands. We can mention some of them such as UNIX [3] original remote login [4], remote shell [5], most modern applications like the ones based on SSH [6], or less standard applications like gexec [7]. But they have some of the next limitations:

- They are platform dependent.
- There is no built-in structure of the host infrastructure we want to administrate.
- Existing tools provide the functionality to execute a command remotely in a single host, but no automation to run scripts in several hosts.
- Existing solutions can only execute shell scripts, which are limited in features, strongly depend on software installed on the destination host, and development in shell script can be tricky, hard and time consuming compared to other programming languages.

With a remote execution system, next actions can be remotely automated:

- Changes to system or application configuration files.
- Host info gathering.
- Software installations.
- Data transfers.
- Backup management.
- Monitoring checks.

Some other programs exists for configuration management, like CfEngine [8], for monitoring, like Nagios [9], for performance statistics like Ganglia [10], but are really focused on their core feature, and does not provide a flexible and integrated way for remote executing complex actions.

Objectives

The main goal of this project is to provide an application that eases all the management tasks associated to a set of hosts.

To achieve this, the application needs to follow next requirements:

- Be secure.
- Be light (use minimal resources on all hosts, and use as few dependencies as possible).
- Keep track of task execution.
- Make user work efficient (do not require users to perform repetitive tasks).
- Be reliable.
- Be powerful (tasks should be able to access file systems, execute operating system commands, access hardware information, etc).

Planning

Time and cost estimation

While some algorithms like COCOMO [11] exist for time and cost estimation, they are probably not very suitable for this specific case. Being Python a very special language in terms of programming efficiency, any method not considering it specifically won't be very accurate. This is specially true, as most of the algorithms are designed with Java in mind, which is much more verbose that Python [16]. Also, it has to be considered that the project will be developed by a single person, in a non-business but academical environment.

So, this estimation does not follow a specific method, and it is rather based on personal experience more than in any existing method.

For cost estimation we will consider a simple model where all hours are rated the same value. Based on the information from Ohloh [17], the average yearly salary for a open source developer is American dollars (USD) 55,000. Counting 11 working months per year, we consider a monthly salary of USD 5,000.

Estimated time dedication for this project is 320 hours. This is equivalent to 2 month work in a full-time basis. So, total development cost for the project is estimated as USD 10,000. We can approximate this value to 7,200 €, setting the price per hour in 22,50 €.

Table 1. Time and cost estimation for the project development

Task	Duration	Cost
Analysis	30 h	675.00 €
Design	50 h	1,125.00 €
Implementation	140 h	3,150.00 €
Task Execution	25 h	562.50 €
Node Communication	40 h	900.00 €
Signature and verification	15 h	337.50 €
Command line utility	30 h	675.00 €
Other	30 h	675.00 €
Packaging	20 h	450.00 €
Testing	50 h	1,125.00 €
Documentation	30 h	675.00 €
TOTAL	**320 h**	**7,200 €**

Gantt chart for the project

Next we define a planning for the tasks of the project formatted as a Gantt chart.

Figure 1.1 Gantt chart for the project development (Feb 28th to Apr 25th)

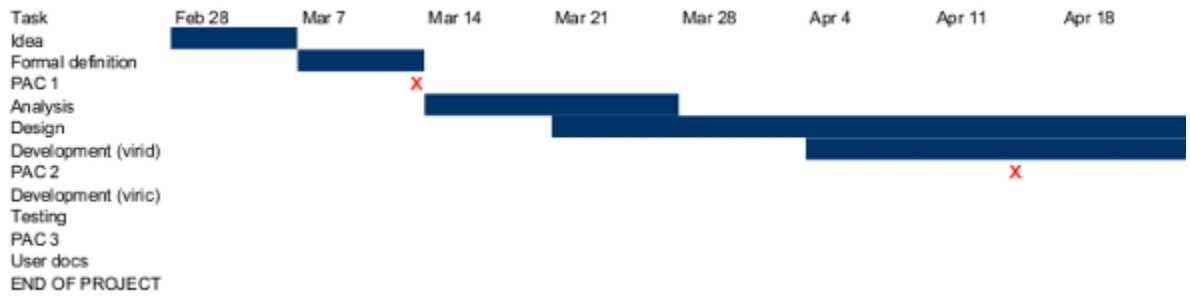
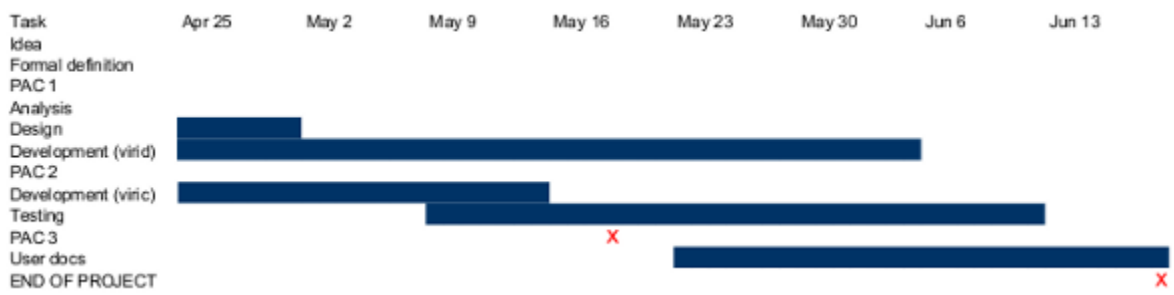


Figure 1.2 Gantt chart for the project development (Apr 25th to Jun 19th)



For presentation reasons we split the chart in two images, but the whole chart is provided, so a global idea of the planning can be better achieved.

Figure 1.3 Gantt chart for the project development (Feb 28th to Jun 19th)

Application design

The goal of the project is to allow a system administrator to automation script execution on a set of remote hosts. To satisfy this general requirement, there are a set of specific requirements that need to be satisfied.

Functional requirements

User requirements

1. Application users must be able to easily send scripts and required data files to the remote hosts.

2. Users must be able to request immediate or scheduled execution of a script.

3. Users must be able to get information about Viri scripts and data files available on the remote hosts, as well as to get information about executed scripts.

Non-functional requirements

Security requirements

1. All communication among the client of the application, and the daemons installed on the remote hosts must be encrypted, not letting any attacker intercept this information in a readable way.

2. Connections to daemons must be only allowed to authenticated clients, not letting unknown users to interact with instances of the application.

Reliability requirements

1. Daemons must be able to capture any exception produced during script execution, and register it, and send it to the user, providing detailed information about the exception.

2. Failures on scripts, or on any external entity which can be controlled, cannot make the daemon crash. Even if the daemon is not able to run normally, it should try to keep alive, and not require a system administrator to manually restore its execution unless it is a must.

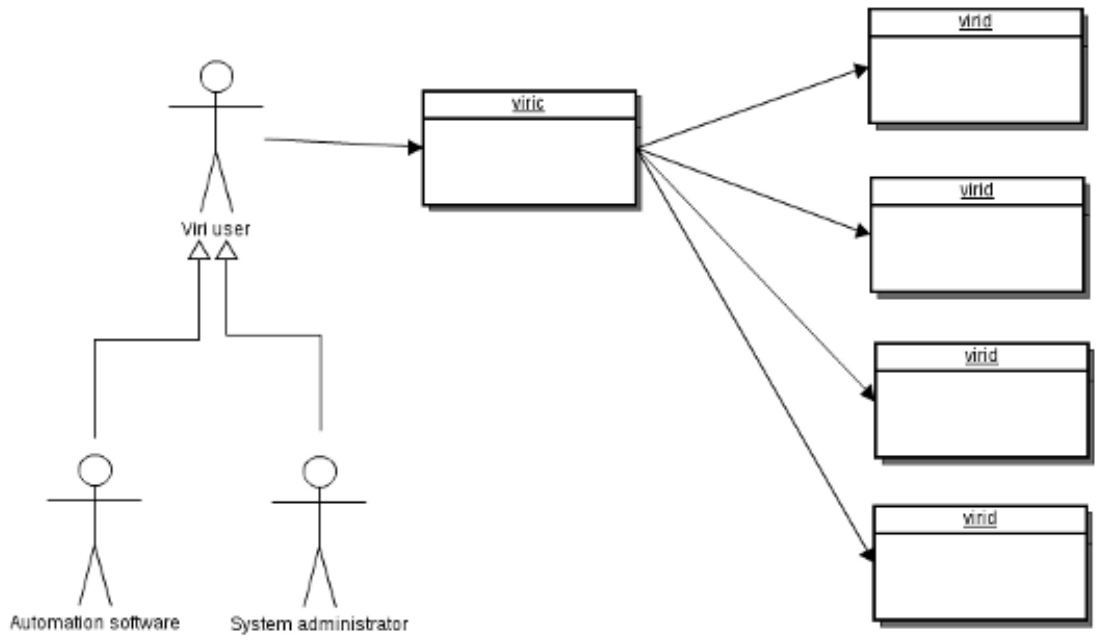
3. Errors on user inputs must be captured and notified back to the user, with useful information to let the user send the correct inputs when retrying.

Actors

Viri consists in two different tools, virid and viric. First one needs to be installed in every host that we want to administrate with Viri. The latter is a command line utility which can be used directly by system administrators, or it can be used to integrate an external application to use the Viri infrastructure.

Next we have a schema representing Viri actors and how they interact.

Figure 2. Schema of Viri actors



System administrator

It is a human user of the application. It is the actor taking the decisions of what to execute, and where it should be executed. It is also responsible for coding the scripts that need to be executed in case they are not in the Viri script library.

Automation software

We can have third party applications that interact with Viri. Imagine our organization have a software to manage all operating system passwords of organization hosts. We can make this application interact with Viri, and make it update operating system passwords every time they are changed in the application.

viric

It is Viri's command line utility. To send orders and data to virid instances it is necessary to use specific protocols which are not human friendly. The idea behind viric is to ease the communication with virid daemons, implementing all the necessary logic to interact with virid, and to provide an easy-to-use user interface.

virid

This is the core component of the Viri application. It is a daemon which is listening on a TCP port, waiting for orders and data from an external entity (usually a viric instance). Also, it optionally has a definition of periodic tasks, and it is checking if any task on this definition is scheduled to run in the current time. In any case a script is required to be executed (due to external order, or to scheduling requirement), virid is responsible of not only execute it, but to keep track of possible errors, to save a history of executions (including what was executed, but also who executed it and at what time).

Use cases

In this section all use cases of the application are defined.

Send script

- Goal: To make the script code available on a remote host.
- Actors: System administrator/Automation software, viric, virid.
- Prerequisites: Script must be coded and available on the local host, and user must be authenticated.

Execute script

- Goal: To perform an action on the remote host. An action can be a maintenance task, a request of information, or anything else that can be coded in the Python language.
- Actors: System administrator/Automation software, viric, virid.
- Prerequisites: Script must exist on the remote host, and user must be authenticated.

List available scripts

- Goal: To let the user know which scripts are available on the remote host, so the user can identify if the one that needs to be executed needs to be sent or not.
- Actors: System administrator/Automation software, viric, virid.
- Prerequisites: User must be authenticated.

List execution history

- Goal: To let the user gather information related to script execution on remote host, and be able to determine if a script needs to be executed, the status of a host, or to take any decision which requires knowledge of the execution history.
- Actors: System administrator/Automation software, viric, virid.
- Prerequisites: User must be authenticated.

Send data file

- Goal: To make a data file available on a remote host, to be used by a Viri script.
- Actors: System administrator/Automation software, viric, virid.
- Prerequisites: Data file must exist in the local host, and user must be authenticated.

Download data file

- Goal: To download from the remote host to the local one, a data file of the Viri application, which can be previously sent by a Viri user, or which has been generated by a Viri script.
- Actors: System administrator/Automation software, viric, virid.
- Prerequisites: Data file must exist in the remote host, and user must be authenticated.

List data files

- Goal: To let the user know which data files are available on the server. This can be useful to know if the data file needs to be sent to the remote host, to be available for a script, or to know which file has been generated by the scripts.
- Actors: System administrator/Automation software, viric, virid.
- Prerequisites: User must be authenticated.

Print data file content

- Goal: This is similar to the download data file use case. In this case, instead of downloading the data file to the local file system, the content is presented on the screen.
- Actors: System administrator/Automation software, viric, virid.
- Prerequisites: Data file must exist in the remote host, and user must be authenticated.

Remove data file

- Goal: To let the user delete a data file from the remote host file system, because it is not needed anymore.
- Actors: System administrator/Automation software, viric, virid.
- Prerequisites: Data file must exist in the remote host, and user must be authenticated.

Rename data file

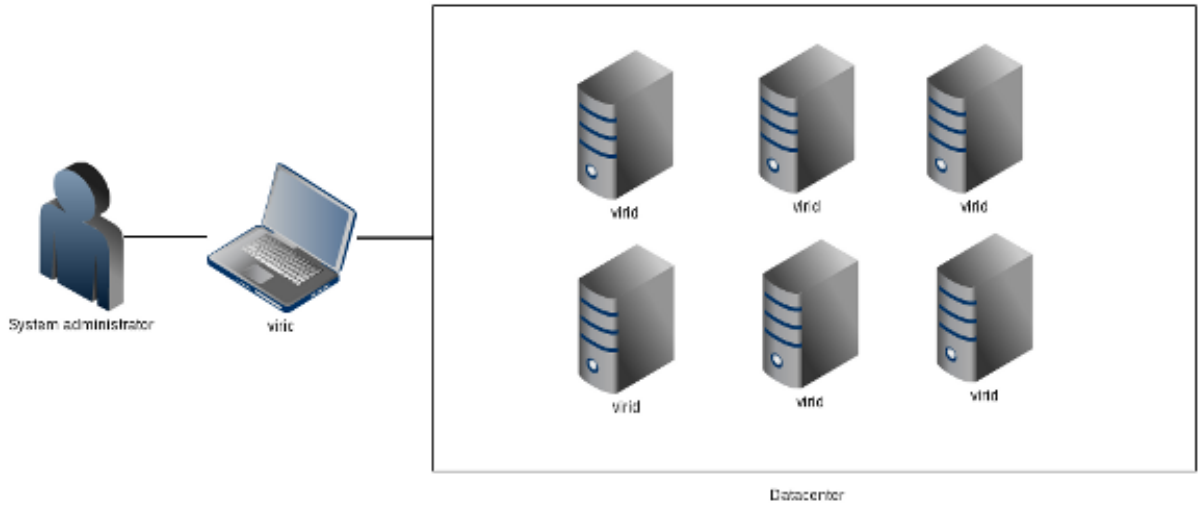
- Goal: To let the user change the name of a data file on the remote host file system.
- Actors: System administrator/Automation software, viric, virid.
- Prerequisites: Data file must exist in the remote host, and user must be authenticated.

Architecture

While Viri can be even used in a single computer, the main idea is to use it to administrate large sets of computers, for example a data center.

Next figure illustrates architecture of Viri in this most common case.

Figure 3. Architecture of Viri, represented on a data center infrastructure



Classes

Packages

Viri is written in Python, which means that it is not structured in packages, in the same way as other programming languages like Java. Nevertheless, the code is structured in directories, which can also be considered packages.

The viri application consists of two different programs, viric and virid. viric is a stand-alone script, so it does not make sense to consider its packages. In the other hand, virid is implemented in different files, in two different directories, so we will consider them the virid packages, as shown in the next diagram.

Figure 4. Package diagram

The reason for having two separate packages is to follow POSIX and Python standards for file organization. The callable virid is placed under a directory named bin. It can be /usr/bin, /usr/local/bin, /opt/bin, etc, depending on the specific system, and its administrator.

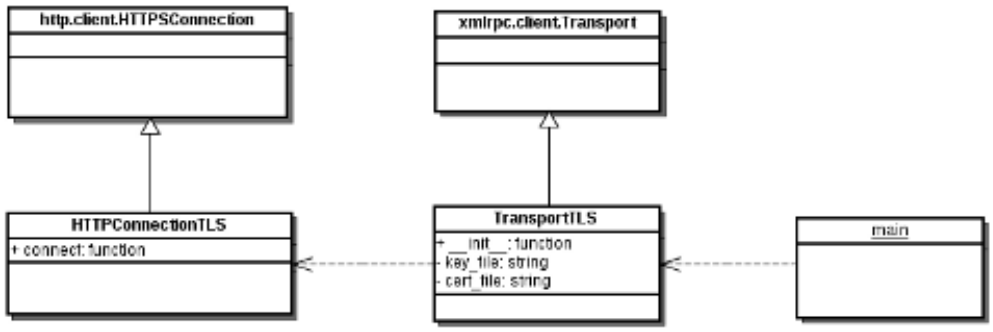
Libraries used by this callable (and actually, they could be reusable by other programs), are placed in Python libraries directory, usually /usr/lib/python/site-packages).

viric classes

The command line utility of the Viri application uses next classes.

- main: Main function of the program, responsible of parsing user provided data, establishing the connection, and calling the remote method.
- xmlrpc.client.Transport: Class from Python standard library which represents the transport layer of the connection to an XML-RPC server.
- TransportTLS: Child class of the previous one, which uses HTTPConnectioTLS to force the use of TLS, and it also implements client authentication on TLS negotiation.
- http.client.HTTPSConnection: Class from Python standard library which implements the connection to an HTTPS server.
- HTTPConnectionTLS: Child of the previous class, that forces the use of TLS over other older protocols like SSLv1 and SSLv2.

Figure 5. Class diagram for viric



Next there is a summary of viric code documentation generated with pydoc:

```
NAME
    viric


CLASSES
    builtins.Exception(builtins.BaseException)
        ConfError
    http.client.HTTPSConnection(http.client.HTTPConnection)
        HTTPConnectionTLS
    xmlrpc.client.Transport(builtins.object)
        TransportTLS


    class ConfError(builtins.Exception)
     |  Represents a error in user provided configuration


    class HTTPConnectionTLS(http.client.HTTPSConnection)
     |  Extending http.client.HTTPSConnection class, so we can specify which
     |  protocol we want to use (we'll use TLS instead SSL)
     |
     |  Method resolution order:
     |      HTTPConnectionTLS
     |      http.client.HTTPSConnection
     |      http.client.HTTPConnection
     |      builtins.object
     |
     |  Methods defined here:
     |
     |  connect(self)


    class TransportTLS(xmlrpc.client.Transport)
     |  Extending xmlrpc.client.Transport class, so we can specify client
     |  certificates needed for client authentication.
     |
     |  Method resolution order:
     |      TransportTLS
     |      xmlrpc.client.Transport
     |      builtins.object
     |
```

```
     |  Methods defined here:
     |
     |  __init__(self, key_file, cert_file, *args, **kwargs)
     |
     |  send_request(self, host, handler, request_body, debug)


FUNCTIONS
    main(cmd, kwargs)
        Handles the connection to the remote virid server, and the execution of
        remote methods. All logic is implemented on the server, so if a invalid
        command is specified, the connection will be established. This makes this
        client generic, and changes to it are rarely required.


    print_error(msg, cmd=None)


    print_help(program)


DATA
    APP_DESC = 'Performs operations on remote hosts using Viri'
    APP_VERSION = '0.1'
    CMD_DEF = {'execute': {'args': (('file_or_id', True),), 'desc': 'Execu...
    DEFAULT_PORT = 6808
    OPT_DATA = (('-d', '--data'), {'action': 'store_true', 'default': Fals...
    OPT_OVERWRITE = (('-o', '--overwrite'), {'action': 'store_true', 'defa...
    OPT_USE_ID = (('-i', '--use-id'), {'action': 'store_true', 'default': ...
    OPT_VERBOSE = (('-v', '--verbose'), {'action': 'store_true', 'default'...
    PROTOCOL = 3


FILE
    bin/viric
```

virid classes

The Viri daemon implements next classes.

- ViriDaemon: This is the main class of the daemon. It takes care of initializing the environment, and it starts both the XML-RPC server and the Scheduler server.

- ScriptManager: Implements all the logic for handling scripts. This includes how scripts are saved, how are listed, how they are executed, etc. This class is used by both the XML-RPC server and the Scheduler server.

- RPCServer: This class implements the XML-RPC server, which allows a remote user to call some specific methods. These methods implement features such as sending scripts, executing them, sending data files, renaming or removing them, etc.

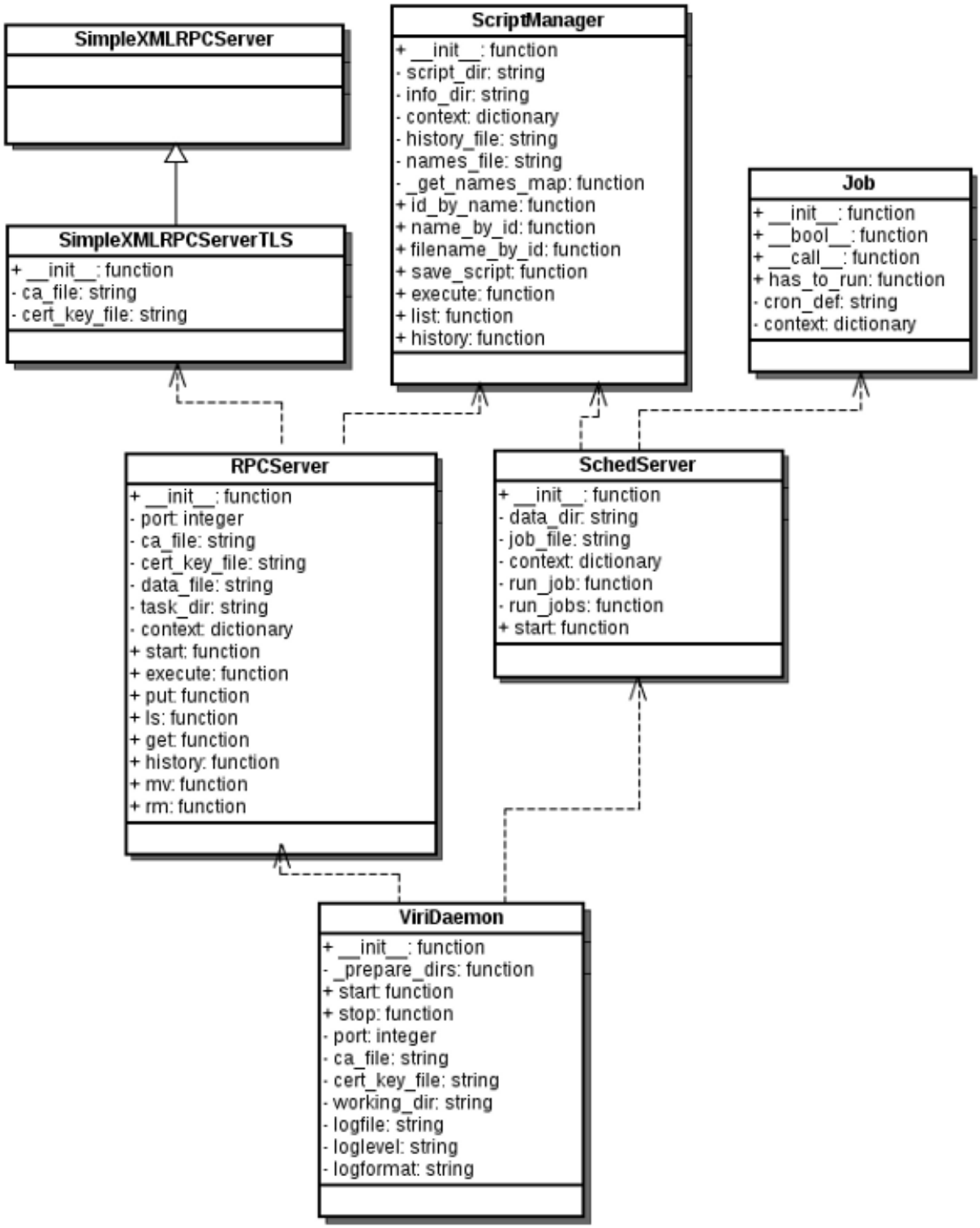
- xmlrpc.server.SimpleXMLRPCServer: Class from Python standard library which implements a simple XML-RPC server.

- SimpleXMLRPCServerTLS: Child of the previous class which works over the TLS protocol, to encrypt the connection, and to authenticate the client.

- SchedServer: This class implements a scheduler server similar to Cron. Every minute it checks a file with the definition of scheduled tasks, and runs specified scripts if the definition specifies that it has to run.

- Job: Class that implements the logic of a single scheduling definition, as specified in previous class.

Figure 6. Class diagram for virid



Next there is a summary of code documentation for virid and its libraries:

```
NAME
    virid


CLASSES
    builtins.object
        ViriDaemon


    class ViriDaemon(builtins.object)
     |  Methods defined here:
     |
     |  __init__(self, config_file)
     |      Initializes all required attributes, getting the values from the
     |      configuration file.
     |
     |      Arguments:
     |      config_file -- path to the configuration file
     |
     |  start(self)
     |      Starts the ViriDaemon. It starts the SchedServer for task
     |      scheduling, and the RPCServer to accept connections form viric
     |      instances
     |
     |  stop(self)



DATA
    APP_DESC = 'Receives and executes scripts from viric instances'
    APP_NAME = 'virid'
    APP_VERSION = '0.0.1'
    DATA_DIR = 'data'
    DEFAULTS = {'General': {'Port': '6806'}, 'Logging': {'LogFile': '/var/...
    DEFAULT_CONFIG_FILE = '/etc/viri/virid.conf'
    INFO_DIR = 'info'
    LOG_LEVELS = ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')
    SCRIPT_DIR = 'script'


FILE
    bin/virid
```

```
NAME
    rpcserver


CLASSES
    builtins.object
        RPCServer
    xmlrpc.server.SimpleXMLRPCServer(socketserver.TCPServer,
xmlrpc.server.SimpleXMLRPCDispatcher)
        SimpleXMLRPCServerTLS


    class RPCServer(builtins.object)
     |  XML-RPC server, implementing the main functionality of the application.
     |
     |  Methods defined here:
     |
     |  __init__(self, port, ca_file, cert_key_file, script_dir, data_dir, script_manager)
     |      Saves arguments as class attributes and prepares
     |      task and data directories
     |
     |      Arguments:
     |      port -- port number where server will be listening
     |      ca_file -- Recognized CA certificates
     |      cert_key_file -- File with daemon's certificate and private key,
     |          for TLS negotiation
     |      script_dir -- directory to store python code representing tasks
     |      data_dir -- directory to store non-code sent files
     |      script_manager -- ScriptManager instance used to handle script
     |          operations
     |
     |  execute = inner(self, kwargs)
     |
     |  get = inner(self, kwargs)
     |
     |  history = inner(self, kwargs)
     |
     |  ls = inner(self, kwargs)
     |
     |  mv = inner(self, kwargs)
     |
     |  put = inner(self, kwargs)
     |
     |  rm = inner(self, kwargs)
     |
```

```
      |  start(self)
      |      Starts the XML-RPC server, and registers all public methods.
      |

    class SimpleXMLRPCServerTLS(xmlrpc.server.SimpleXMLRPCServer)
      |  Overriding standard xmlrpc.server.SimpleXMLRPCServer to run over TLS.
      |  Changes inspired by
      |  http://www.cs.technion.ac.il/~danken/SecureXMLRPCServer.py
      |
      |  Method resolution order:
      |      SimpleXMLRPCServerTLS
      |      xmlrpc.server.SimpleXMLRPCServer
      |      socketserver.TCPServer
      |      socketserver.BaseServer
      |      xmlrpc.server.SimpleXMLRPCDispatcher
      |      builtins.object
      |
      |  Methods defined here:
      |
      |  __init__(self, addr, ca_file, cert_key_file, requestHandler=<class
'xmlrpc.server.SimpleXMLRPCRequestHandler'>, logRequests=True, allow_none=False,
encoding=None, bind_and_activate=True)
      |      Overriding __init__ method of the SimpleXMLRPCServer
      |
      |      The method is a copy, except for the TCPServer __init__
      |      call, which is rewritten using TLS, and the certfile argument
      |      which is required for TLS


FUNCTIONS
    protect(directory, filename)
        Returns the absolute path to the file in the directory, only if the
        filename is in the directory. This is done to prevent access to files
        outside the Viri working directory, if the user sends as a parameter
        something like ../../../etc/passwd


    public(func)
        Decorator that controls arguments of public methods. XMLRPC only
        works with positional arguments, so the client always sends only one
        parameter with a dictionary. This decorator executes the method
        passing the items in the dictionary as keyword arguments.
        It also captures any error non-controlled error, logs it, and sends
        it to the client as text.
```

```
DATA
    ERROR = 1
    PROTOCOL = 3
    RPC_METHODS = ('execute', 'put', 'ls', 'get', 'history', 'mv', 'rm')
    SUCCESS = 0


FILE
    viri/rpcserver.py
```

```
NAME
    schedserver

CLASSES
    builtins.Exception(builtins.BaseException)
        InvalidCronSyntax
    builtins.object
        Job
        SchedServer

    class InvalidCronSyntax(builtins.Exception)
     |  Error when parsing a cron syntax execution schedule
     |
     |  Method resolution order:
     |      InvalidCronSyntax
     |      builtins.Exception
     |      builtins.BaseException
     |      builtins.object

    class Job(builtins.object)
     |  Represents a scheduled execution of a script
     |
     |  Methods defined here:
     |
     |  __bool__(self)
     |      Specifies if the job was a real job, or a comment, a blank line
     |      or raised an invalid cron syntax. True means it was a valid job.
     |
     |  __call__(self)
     |      Method executed when the job has to run. It executes the task
     |      specified in the cron definition
     |
     |  __init__(self, cron_def, script_manager)
     |
     |  has_to_run(self, now)
     |      Returns a boolean representing if the job has to run in the
     |      current time, specified by now.
```

```
    class SchedServer(builtins.object)
     |  Daemon which simulates the cron application, but instead of executing
     |  shell commands, it executes viri tasks.
     |
     |  Methods defined here:
     |
     |  __init__(self, data_dir, script_manager)
     |      Initializes the SchedServer, by setting the path of the jobs file
     |
     |      Arguments:
     |      data_dir -- directory where data files are stored
     |      script_manager -- ScriptManager instance used to handle script
     |          operations
     |
     |  start(self)
     |      Starts the SchedServer. It gets the current time (date, hour and
     |      minute), and checks for all jobs in the jobs file, if any of them
     |      has to be executed in the current minute, and executes them.
     |
     |      Jobs are called in new threads, so they shouldn't block the execution
     |      process. But in case the scheduling process is delayed more than a
     |      minute, it doesn't skip any minute, and it executes tasks even if they
     |      are delayed.


DATA
    JOBS_FILE = '__crontab__'
    SLEEP_TIME = 5


FILE
    viri/schedserver.py
```

```
NAME
    scriptmanager


CLASSES
    builtins.object
        ScriptManager


    class ScriptManager(builtins.object)
     |  Methods defined here:
     |
     |  __init__(self, script_dir, info_dir, context)
     |      Sets class attributes
     |
     |      Arguments:
     |      script_dir -- directory where scripts are stored
     |      info_dir -- directory where information about scripts is stored
     |      context -- dictionary containing values that will be available
     |          as attributes of the ViriScript class in the script
     |
     |  execute(self, script_id)
     |      Executes the specified script. The script must contain a class
     |      named ViriScript, which implements a run method, containing the entry
     |      point for all the script functionality. This way, we're able to add
     |      some attributes to the class with host specific information.
     |
     |      Example script, returning if data directory has already been created
     |      (note that the data_dir attribute is not defined on the class, but is
     |      added on execution time through the context):
     |
     |      >>> import os
     |      >>>
     |      >>> class ViriScript:
     |      >>>     def run(self):
     |      >>>         return os.path.isdir(self.data_dir)
     |
     |      Arguments:
     |      context -- extra information that will be made available on attributes
     |          of the ViriScript on the script
     |
```

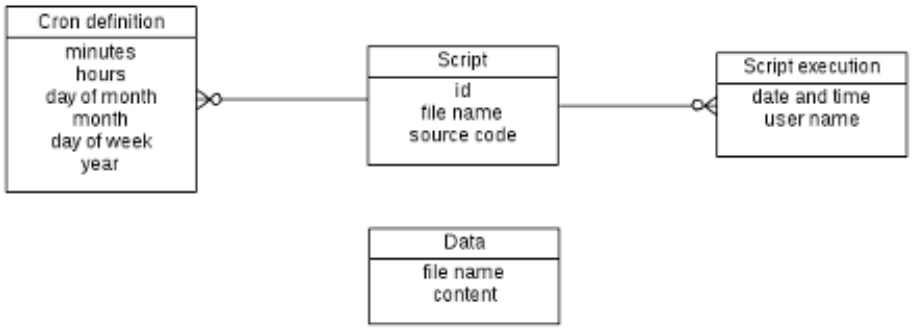
```
      |  filename_by_id(self, script_id)
      |      Returns the absolute path of a script given its id
      |
      |  history(self)
      |
      |  id_by_name(self, script_name)
      |      Returns the id of the last version of a script given its name
      |
      |  list(self, verbose=False)
      |      List all installed scripts
      |
      |  name_by_id(self, script_id)
      |      Returns the name of a script given its id
      |
      |  save_script(self, filename, content)
      |      Saves the script in the scripts directory, adds its id to the
      |      information file of its file name, and adds the id and name to the
      |      names dictionary.
      |
      |      Arguments:
      |      filename -- original script file name
      |      content -- script content (code)
      |
      |  ----------------------------------------------------------------------
      |  Data and other attributes defined here:
      |
      |  ExecutionError = <class 'scriptmanager.ExecutionError'>
      |      Represents any error during script execution


FUNCTIONS
    sha1 = openssl_sha1(...)
        Returns a sha1 hash object; optionally initialized with a string


DATA
    BASE_SCRIPT = '__base__'
    HISTORY_FILE = 'history'
    NAMES_FILE = 'names.json'


FILE
    viri/scriptmanager.py
```

Entity-Relation Diagram (ERD)

While Viri does not use a database to store data, there is some information managed by Viri, and stored in different files. For the viric program, is the user who directly manages files and information. So, with just need to consider the entities and relations of the information managed by virid.

Next, there is the ERD representing the information managed by Viri:

Figure 7. Entity-Relation Diagram of the Viri application



The core of Viri information are the scripts. Scripts are Python code with a specific structure which implement a task to perform on hosts with virid running. Scripts can optionally be scheduled in cron definitions. A script can be scheduled in more than one definition. When a script is executed (independently if it is executed manually, or scheduled), information about the execution is stored. Specifically, the date and time of the execution, and the user name (which is get from the certificate), is saved.

Finally, Viri also manages data files. Data files are just transferred from the user workstation, to the hosts running virid, keeping their names, and making them available to scripts. So, there is no special operation or feature regarding data files.

Used technologies

Python

Python is the programming language used to implement the whole application. Also it is the scripting language used to write the scripts.

From the Python home page “Python is a programming language that lets you work more quickly and integrate your systems more effectively. You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs.”.

This is the main reason why Python is the right language for this project. Python provides a very powerful language, good performance for a scripting language, and at the same time it provides a clear syntax and a huge set of included libraries, which makes coding very efficiency.

Also, Python is Free and Open Source Software [14], and runs in the main operating systems and architectures.

There are two main versions of the Python language, the 2.x and the 3.x series. While Python 2.x is still used for most of the projects and libraries, Viri is written for Python 3.1. This is specially possible because Viri does not depend on any external library. Depending on libraries which are only available for Python 2.x is the main reason why most projects are not written or ported to the newest 3.x versions.

XML-RPC

There are several protocols and technologies for communicating the components of an application. For what Viri has to do, what makes more sense is to use a Remote Procedure Call (RPC) [16] technology. Among the several available options, probably CORBA [17] is the one which performs better [18]. But Python comes with both a client and a server implementation of XML-RPC [19] on the standard library. Using XML-RPC avoids requiring extra dependencies, which keeps the application simpler and smaller. Also, usually Python core packages are better tested and more reliable than third-party libraries.

XML-RPC is a Remote Procedure Call protocol, which represents data in XML [20], and uses HTTP [21] to transfer it.

Transport Layer Security (TLS)

Viri is an application which allows executing a script on remote machines. Because of this, the application is very sensitive to attacks, and it is very important to provide all mechanisms to protect its infrastructure. Without proper security, Viri could be used by attackers to run malicious code on hosts running the virid daemon, and even take control of the system.

There is some security which needs to be performed outside the scope of the application itself, like protecting networks with firewalls, but there are some which is provided by the application itself, and its explained below.

To secure all communications between Viri components, the application encrypts all data which is sent using Transport Layer Security (TLS) version 1 [22]. TLS is also used to authenticate the connections. To do this, it is necessary to implement and use a Public Key Infrastructure (PKI) [23], and use required certificates with the Viri application. Authenticating connections, we can be sure that only authorized people or systems can communicate with the virid daemons.

While Python's XML-RPC module includes some of the necessary features for running over TLS, there are some of them missing that Viri itself implements, like client authentication, or forcing TLS over older and more insecure version of the Secure Sockets Layer (SSL) [24] protocol family.

Python includes an ssl module [25] since version 2.6, which is used by Viri to implement TLS encryption of XML-RPC. This module allows to secure the socket used for the HTTP communication, with SSL or TLS.

Git

Git [28] is the version control system used to manage Viri code.

From the Git site "Git is a free & open source, distributed version control system designed to handle everything from small to very large projects with speed and efficiency.".

Sphinx / reStructuredText

reStructuredText [29] is the *de facto* standard syntax for writing Python documentation. With reStructuredText, documentation is written as plain text, using some special wildcards for formatting (bold and italic letters, headers, etc), as well as adding links and images.

Sphinx [30] is a program which allows creating a reStructuredText project, and to export it to different formats. Viri uses it to generate HTML documentation.

Protocols

Viri is built on top of a set of protocols, which encapsulate data between viric and virid at different levels.

In the next schema, it is represented the protocol structure of Viri:

Table 2. Schema representing Viri's network protocols

Viri
XML-RPC
HTTP
SSL
TCP

In a first step, Viri calls are encapsulated over the XML-RPC protocol. The structure of a XML-RPC call looks like this.

```
<?xml version="1.0"?>
<methodCall>
  <methodName>send_data</methodName>
  <params>
    <param>
        <data_filename><string>httpd.conf</string></data_filename>
    </param>
    <param>
        <data_binary><base64>eW91IGNhbid0IHJlYWQgdGhpcyE=</base64></data_binary>
    </param>
    <param>
        <overwrite><boolean>1<boolean></overwrite>
    </param>

  </params>
</methodCall>
```

User documentation

Getting started

What is Viri?

The aim of the Viri project is to provide an efficient and organized way to run Python scripts in remote hosts. This is specially useful to perform system administration tasks in data centers.

Imagine you are the administrator of 1000 hosts (physical or virtual). With Viri, you can automate tasks to all those hosts almost like if you had to perform them in one. Some examples of tasks you would like to run include:

- Change configuration files.
- Run backups.
- Make all your hosts send system data to a central location, so you have an up-to-date database of all your infrastructure. Sent data could include IP addresses, uptime, listening ports, operating system version, installed packages, available security updates, processor architecture, disk partitions, information found in logs, etc.
- Install software.
- And everything you are able to code in Python.

While providing a set of scripts, the idea behind Viri is to let system administrators to write their own scripts. Viri is just a framework for remotely executing any Python script.

Viri's core is the virid daemon, which needs to be installed in every host of the infrastructure. Then, the command line utility, or a custom application can be used to manage scripts and data.

Installation

The core component of the Viri application is the virid daemon. The daemon is distributed in the next ways:

- A DEB package for GNU/Linux distributions like Debian and Ubuntu.
- A RPM package for GNU/Linux distributions like RedHat and CentOS.
- By source code.

The viric command line utility is a single Python file.

Before using Viri, a Public Key Infrastructure needs to be defined, and the proper certificates need to be installed, in both the daemons and the command line utility instances. See Creating a Public Key Infrastructure (PKI) for Viri section for more information.

To install the virid daemon in a Debian based GNU/Linux distribution, run the next command:

```
sudo dpkg -i virid.deb
```

Then, install required keys, and set their paths (and update any other setting) on /etc/viri/virid.conf

Finally, to start the virid daemon, run:

```
sudo /etc/init.d/virid start
```

viric

This section details all the commands available for the viric command line utility.

Usage

viric can be executed from a shell, with next syntax.

```
./viric --host=<host> <command> [options]
```

Where <command> is on of the commands explained in this sections, and [options] is a set of options, including common options or options specific to used command.

Common options

--help (-h)

- Display usage information.

--version (-v)

- Display program version.

--host (-H)

- Specifies the remote host viric will connect to.

--port (-p)

- Specifies the port where virid is listening in the remote host. Default is 6808.

--keyfile (-p)

- Path to the file where user's private key is located.

--certfile

- Path to the file where user's certificate is located.

Available commands

put <script file name>

- Sends a script from a file in a local filesystem to a remote host, and returns the script id, which is a hash calculated from its source code.

  --data (-d): Sends a data file, instead of a script, to the remote host.

  --overwrite (-o): Overwrite the remote file if it already exists.

execute <script file name>

- Sends and executes a script in the remote host. Local process exit with return code 0 if execution completes successfully, and with a code different than 0 if the script has an execution error. The result of the script is displayed in the local host, or the remote error and its traceback if the script fails.

ls

- Returns the list of all remote scripts with their names, ids and last updated times.

  --data (-d): Lists data files instead of scripts.

history

- Returns the log of all script executions, showing the date and time of the execution, the script name and id, and the common name of the owner of the certificate who executed the script.

get <file name or script id>

- Download a script from the remote host, and save it with the same name on the current directory.

  --data (-d): Download a data file instead of a script.

mv <source file name> <destination file name>

- Renames a file in the remote host working directory.

  --overwrite (-o): Overwrite the destination file if it already exists.

rm <file name>

- Removes a data file from the remote host working directory.

virid configuration

virid configuration file contains a set of directives to customize the behavior of the virid daemon.

Next, there is an explanation of all available directives, which are grouped in sections.

General

This section contains general settings for virid.

Port

- Port the virid daemon will be listening at. This port is used by a viric client, and protocol used is XML-RPC over TLS.

  Default is 6808

Paths

This section contains file paths that are customizable in Viri.

KnownCAs

- Path to a file containing certificates of the certificate authorities (CA) which are known. The only requests accepted by the virid daemon will come from clients with certificates issues (signed) by one of this known CAs.

  Default is /etc/viri/ca.cert

CertKeyFile

- Path to a file containing virid private key and certificate. Certificate can be self signed, as it is not authenticated by viric instances.

  Default is /etc/viri/virid.pem

WorkingDir

- Working directory to be used by the virid daemon to store necessary files, such as scripts and data received from the remote clients.

  Default is /var/viri

Logging

This section describes the settings related to virid logged information.

LogFile

- Path where virid activity will be logged.

  Default is /var/log/virid.log

LogLevel

- Minimum severity of logged messages. Must be one of: DEBUG, INFO, WARNING, ERROR, CRITICAL

  Default is WARNING

LogFormat

- Defines the format of logged messages. See next page for details: http://docs.python.org/release/3.1.3/library/logging.html#formatter-objects

  Default is %(levelname)s::%(asctime)s::%(message)s

Creating a Public Key Infrastructure (PKI) for Viri

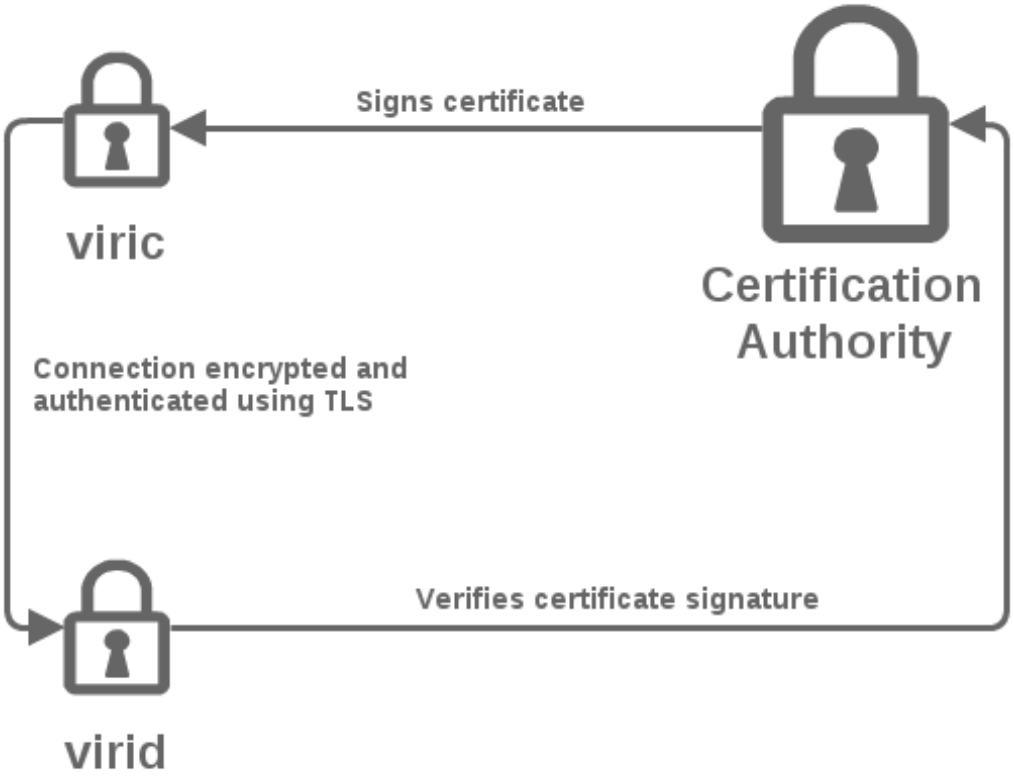
Introduction

The Viri application is a very powerful tool. But at the same time it is very sensitive to attacks. For this reason, security is very important, and Viri is build on the top of a encryption, and certification infrastructure.

Every host running Viri has a daemon waiting for orders. The goal of creating a Public Key Infrastructure (PKI) is to ensure that a) only authorized users or applications can send this orders and its associated data; b) all the communications among Viri components are encrypted and sent information can only be understood by the sender and the receiver.

Next schema shows how Viri security is achieved.

Figure 8. Schema of Viri authentication



With this aim, we will define next elements in the PKI:

- Certification Authorities
- Viri users
- Viri hosts

A Certification Authority is an entity we can trust on.

A Viri user is a person or an application who can send orders to one or more hosts in the infrastructure.

A Viri host is a physical or virtual machine running an operating system, and running the Viri daemon.

Certification Authorities

A Certification Authority (CA) is the root node of a trust network. The only responsibility of the CA is to validate the identity of all the other elements in the PKI.

We can define more than one CA for a Viri infrastructure, and the CA can be internal or external to our organization. Basically a CA is an entity which keeps a private key in private, and signs certificates with it. The CA has a public certificate itself, that is used to validate the signatures of the certificates.

So, it can be considered that the CA is just an entity we trust on. And we trust that it will only sign the certificates of authorized users of the system.

There are several existing companies we can use as CA, such as CACert.org, Thawte, Verisign, Positive SSL, Equifax and GeoTrust. But in this document we will show how to create a custom CA.

First thing for creating a our own CA is to create its private key (which must be kept in secret), and its certificate (which will be public).

To achieve this, we will use the OpenSSL command for the GNU/Linux operating system. Next command will create a private key and a self-signed certificate, valid for 10 years:

```
openssl req -x509 -nodes -newkey rsa:4096 -keyout ca.key -out ca.cert -days 3650
```

This command will generate two different files:

- ca.key: The private key
- ca.cert The public certificate

The private key will be kept in a secure place, and it will only be used to sign the certificates of the authorized users.

The public certificate will be distributed to any entity which needs to validate user certificates, which is every instance of the virid daemon.

Then, we will initialize a working environment for our Certification Authority. Next steps create two files required by OpenSSL in order to perform CA tasks.

```
mkdir demoCA
touch demoCA/index.txt
echo "00" > demoCA/serial
```

First we create the index.txt file, where a log of signed certificates will be saved. Second, we create and initialize a serial number for signed certificates.

Creating a real Certification Authority can require a more complex set up, and more knowledge of the procedure, so it is recommended to read some specific documentation on this topic if a custom CA is going to be used.

Viri users

We can define a Viri user as any entity who can establish a connection with virid daemons. This can be a system administrator, or also an application.

Because virid clients need to be authenticated, every Viri user needs a pair of keys, and a certificate issued by one of the Certification Authorities. This way, if we trust the Certification Authority, we can trust the user is who she claims to be.

First thing we need to do to be able to authenticate a Viri user is to generate its private key. We can do it with next command:

```
openssl genrsa 4096 > new_user.key
```

Then, we need to generate a Certificate Signing Request (CSR), which basically is a certificate ready to be signed by a Certification Authority. To generate it, we can proceed with next command:

```
openssl req -new -key new_user.key -out new_user.csr
```

And provide required information such as country, city, organization, common name, etc.

Finally, from the CA directory created in the previous section, we will sign the CSR and generate the certificate for the user.

```
openssl ca -in new_user.csr -cert ca.cert -keyfile ca.key -policy policy_anything -outdir .
        -out new_user.cert
```

Now, we have the two files required by the Viri client viric to be able to connect to Viri daemons.

Viri hosts

A Viri host is any machine (virtual or physical) which runs the virid daemon. The virid daemon is listening on a port and waiting for orders, but we want to accept orders only from authorized users. Actually, virid does not even accept connections from users that are not authenticated.

To define which users are authorized, virid relies on information from Certification Authorities. So, when installing virid, a file with known CAs needs to be provided and configured correctly.

Then, when a client tries to connect with the virid daemon, it checks if this client is providing in the connection a certificate issues by one of the known CAs, and it will only accept the connection if this is the case.

To establish connections with the clients, Viri host also need a key and certificate pair. In this case, the clients want authenticate the certificate using a Certification Authority, so it is possible to create a self-signed certificate. Next command will create a self-digned certificate using OpenSSL:

```
openssl req -new -newkey rsa:4096 -nodes -x509 -keyout new_host.pem -out new_host.pem
        -days 3650
```

ViriScript

Viri is a system to execute scripts, so we need to define how a Viri script looks like.

To let Viri to have a better control on the execution, and to provide additional context available in the script, Viri scripts must define a class ViriScript. This class needs to define a run method too, which will be called by virid when a client requests the script execution. The return of this method is what will be returned back to the client.

Here there is an example of a simple Viri script.

```
class ViriScript:
    def run(self):
        import datetime
        return 'Host time: %s, Data dir: %s' % (datetime.datetime.now(), self.data_dir)
```

This script just gets the date and time on the host, and returns it together with the path to Viri's data directory. Python's datetime module is used to know the date and time in the host, and Viri's data directory is an attribute of the ViriScript added by the virid daemon, so the programmer of the script can access files in this directory.

Scheduled tasks

Viri can not only execute scripts when a client requests to, but also on a specific date and time, or periodically following a pattern of executions.

To schedule Viri script executions, a special file is provided, __crontab__. This file follows the exact same syntax as the crontab file of the UNIX cron application with one exception, instead of specifying the command to run it needs to specify the script id to execute.

Next table define the meaning of every field (space separated) in the __crontab__ file.

Table 3. __crontab__ field reference

Field
Minute
Hour
Day of month
Month
Day of week
Year (optional)
Script id

Next, there is a sample __crontab__ file, that would make the script with id 99154c826fca745be859c6481a5f87631e4b2b78, to be executed every hour, at the 30 minutes, and also on December 31st, 2015, at 23:59.

```
# Every hour, at the half past hour
30 * * * * 99154c826fca745be859c6481a5f87631e4b2b78


# On December 31st, 2015 at 23:59
59 23 31 12 * 2015 99154c826fca745be859c6481a5f87631e4b2b78
```

Base script

Viri brings a way to define a base script for all defined scripts. A base script is a ViriScript class all ViriScript classes inherit from.

Imagine we want to implement a common feature in all our scripts, for example, we want all the Viri scripts to access a web service we want to use to stop all script executions for some periods of time. Implementing the same code in every Viri script we develop is not very efficient, so we can use a base script where we are going to define this.

```
class ViriScript:
    def wait_until_allowed(self):
        import urllib
        import time
        while True:
            if urllib.urlopen('http://localhost:8000') == 'ok':
                break
            time.sleep(60)
```

To make the base script available in a Viri host, it just needs to be sent to the host with the name __base__.py using the send command of viric.

Then, when developing our scripts, we can use this method, because even if it is not explicitly defined, our script class will inherit from our base class.

```
class ViriScript:
    def run(self):
        import datetime
        self.wait_until_allowed()
        return 'Host time: %s, Data dir: %s' % (
            datetime.datetime.now(), self.data_dir)
```

Appendix

Patching Python's XML-RPC to use TLS

Python provides an implementation of both XML-RPC client and server which are used by this project.

While this library works for more regular usage of a XML-RPC server, Viri requires extra features that are not implemented. These features include running the XML-RPC server over TLS, forcing the use of TLS over SSL on the client, and allowing client authentication in both sides.

This changes should be implemented on Python itself, but being this outside the scope of this project, child classes of the original Python ones are created, reusing all original feature, and adding required one.

Next, there is the code of this classes, for both, the server and the client (in this order).

```
class SimpleXMLRPCServerTLS(SimpleXMLRPCServer):
    """Overriding standard xmlrpc.server.SimpleXMLRPCServer to run over TLS.
    Changes inspired by
    http://www.cs.technion.ac.il/~danken/SecureXMLRPCServer.py
    """
    def __init__(self, addr, ca_file,
        requestHandler=SimpleXMLRPCRequestHandler, logRequests=True,
        allow_none=False, encoding=None, bind_and_activate=True):
        """Overriding __init__ method of the SimpleXMLRPCServer

        The method is a copy, except for the TCPServer __init__
        call, which is rewritten using TLS, and the certfile argument
        which is required for TLS
        """
        self.logRequests = logRequests
        SimpleXMLRPCDispatcher.__init__(self, allow_none, encoding)
        socketserver.BaseServer.__init__(self, addr, requestHandler)
        self.socket = ssl.wrap_socket(
            socket.socket(self.address_family, self.socket_type),
            server_side=True,
            certfile='keys/virid.pem', # FIXME set as an argument
            ca_certs=ca_file,
            cert_reqs=ssl.CERT_REQUIRED,
            ssl_version=PROTOCOL,
            )
        if bind_and_activate:
            self.server_bind()
            self.server_activate()

        # [Bug #1222790] If possible, set close-on-exec flag; if a
        # method spawns a subprocess, the subprocess shouldn't have
        # the listening socket open.
        try:
            import fcntl
        except ImportError:
            pass
        else:
            if hasattr(fcntl, 'FD_CLOEXEC'):
                flags = fcntl.fcntl(self.fileno(), fcntl.F_GETFD)
                flags |= fcntl.FD_CLOEXEC
                fcntl.fcntl(self.fileno(), fcntl.F_SETFD, flags)
```

```
class HTTPConnectionTLS(http.client.HTTPSConnection):
    """Extending http.client.HTTPSConnection class, so we can specify which
    protocol we want to use (we'll use TLS instead SSL)
    """
    def connect(self):
        sock = socket.create_connection((self.host, self.port), self.timeout)
        if self._tunnel_host:
            self.sock = sock
            self._tunnel()
        self.sock = ssl.wrap_socket(sock, self.key_file, self.cert_file,
            ssl_version=PROTOCOL)



class TransportTLS(xmlrpc.client.Transport):
    """Extending xmlrpc.client.Transport class, so we can specify client
    certificates needed for client authentication.
    """
    def __init__(self, key_file, cert_file, *args, **kwargs):
        self.key_file = key_file
        self.cert_file = cert_file
        super(TransportTLS, self).__init__(*args, **kwargs)

    def send_request(self, host, handler, request_body, debug):
        host, extra_headers, x509 = self.get_host_info(host)
        connection = HTTPConnectionTLS(
            host,
            None,
            self.key_file,
            self.cert_file,
            **(x509 or {}))
        if debug:
            connection.set_debuglevel(1)
        headers = {}
        if extra_headers:
            for key, val in extra_headers:
                headers[key] = val
        headers['Content-Type'] = 'text/xml'
        headers['User-Agent'] = self.user_agent
        connection.request('POST', handler, request_body, headers)
        return connection
```

Unit Tests

Viri includes a set of unit tests, which provides next functions:

- Verify that code behaves as expected.
- Provide a way to verify that future changes does not break current features.
- Document usage of code units in a practical way.

Next, there is the definition of all provided tests:

```
# rpcserver.py tests

r"""
>>> import shutil
>>> import tempfile
>>> import xmlrpc.client
>>> from viri import scriptmanager
>>> from viri import rpcserver

# Dummy initialization of the server because we are going to access the
# methods directly, and not using the XML-RPC server. We are only
# interested on providing known values for directories
>>> script_dir = tempfile.mkdtemp(prefix='viri-tests-scripts-')
>>> data_dir = tempfile.mkdtemp(prefix='viri-tests-data-')
>>> info_dir = tempfile.mkdtemp(prefix='viri-tests-info-')
>>> context = {}
>>> script_manager = scriptmanager.ScriptManager(
...     script_dir, info_dir, context)
>>> rpcs = rpcserver.RPCServer(6808, '', '', script_dir, data_dir,
...     script_manager)
```

```
##################################
### put and get of data files ###
##################################

>>> rpcs.put({'file_name': 'filename1',
...     'file_content': xmlrpc.client.Binary(b'filaname1 content\n'),
...     'data': True})
(0, 'Data file filename1 saved')
>>> rpcs.put({'file_name': 'filename2.txt',
...     'file_content': xmlrpc.client.Binary(b'filaname2 content\n'),
...     'data': True})
(0, 'Data file filename2.txt saved')
>>> rpcs.put({'file_name': 'filename3.py',
...     'file_content': xmlrpc.client.Binary(b'print("hello world!")\n'),
...     'data': True})
(0, 'Data file filename3.py saved')
>>> rpcs.get({'filename_or_id': 'filename1', 'data': True})[1].data
b'filaname1 content\n'
>>> rpcs.put({'file_name': 'filename1',
...     'file_content': xmlrpc.client.Binary(b'filaname1 content modified\n'),
...     'data': True})
(0, 'Not overwriting file filename1')
>>> rpcs.get({'filename_or_id': 'filename1', 'data': True})[1].data
b'filaname1 content\n'
>>> rpcs.put({'file_name': 'filename1',
...     'file_content': xmlrpc.client.Binary(b'filaname1 content modified\n'),
...     'data': True, 'overwrite': True})
(0, 'Data file filename1 overwrote')
>>> rpcs.get({'filename_or_id': 'filename1', 'data': True})[1].data
b'filaname1 content modified\n'
>>> ls_res = rpcs.ls({'data': True})[1].split('\n')
>>> ls_res.sort()
>>> ls_res
['filename1', 'filename2.txt', 'filename3.py']
```

```
##################################
### operations with data files ###
##################################

>>> rpcs.mv({'source': 'filename1', 'destination': 'filename4'})
(0, 'File filename1 successfully renamed to filename4')
>>> rpcs.mv({'source': 'filename1', 'destination': 'filename4'})
(1, 'File filename1 not found')
>>> rpcs.mv({'source': 'filename4', 'destination': 'filename2.txt'})
(1, 'Rename aborted because destination file already exists. Use --overwrite to force it')
>>> rpcs.mv({'source': 'filename4', 'destination': 'filename2.txt', 'overwrite': True})
(0, 'Existing file filename2.txt replaced by renaming filename4')
>>> rpcs.rm({'filename': 'filename2.txt'})
(0, 'File filename2.txt successfully removed')
>>> rpcs.rm({'filename': 'filename2.txt'})
(1, 'File not found. File names cannot include directories')
>>> ls_res = rpcs.ls({'data': True})[1].split('\n')
>>> ls_res.sort()
>>> ls_res
['filename3.py']

##############################
### put and get of scripts ###
##############################

>>> rpcs.get({'filename_or_id': 'script1.py',})
(1, 'File not found. File names cannot include directories')
>>> rpcs.put({'file_name': 'script1.py',
...     'file_content': xmlrpc.client.Binary(b'print("hello world!")\n')})
(0, '5db8eb03071c8c231a8f51b3d2a98bd1eb589634')
>>> rpcs.get({'filename_or_id': 'script1.py',})[1].data
b'print("hello world!")\n'
>>> rpcs.put({'file_name': 'script1.py',
...     'file_content': xmlrpc.client.Binary(b'print("hello viri!")\n')})
(0, 'e93c885fc9865db7369d9b516e839a789e8a1622')
```

```
##########################
### script definitions ###
##########################

>>> BASE_CORRECT = '''
class ViriScript:
    base_attr = True
'''
>>> SCRIPT_CORRECT = '''
class ViriScript:
    def run(self):
        return '__base__ used: %s ' % hasattr(self, 'base_attr')
'''
>>> SCRIPT_NO_CLASS = '''
base_attr = True
'''
>>> SCRIPT_SYNTAX_ERROR = '''
print('syntax error
'''


####################################
### script execution and history ###
####################################

>>> rpcs.history({})
(0, '')
>>> rpcs.execute({'script_id': 'e93c885fc9865db7369d9b516e839a789e8a1622', 'use_id': True})
(1, '')

################
### clean up ###
################

>>> shutil.rmtree(script_dir)
>>> shutil.rmtree(data_dir)
>>> shutil.rmtree(info_dir)

"""

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

```
# schedserver.py tests

r"""
>>> import datetime
>>> import tempfile
>>> import shutil
>>> from viri import scriptmanager
>>> from viri import schedserver

>>> script_dir = tempfile.mkdtemp(prefix='viri-tests-scripts-')
>>> data_dir = tempfile.mkdtemp(prefix='viri-tests-data-')
>>> info_dir = tempfile.mkdtemp(prefix='viri-tests-info-')
>>> context = {}
>>> script_manager = scriptmanager.ScriptManager(
...     script_dir, info_dir, context)

###########
### Job ###
###########

# Execute every minute
>>> job = schedserver.Job('* * * * * hash', script_manager)
>>> job.has_to_run(datetime.datetime(2011, 4, 30, 23, 23, 12))
True
>>> job.has_to_run(datetime.datetime(1999, 12, 31, 23, 59, 59))
True
>>> job.has_to_run(datetime.datetime(2000, 1, 1, 0, 0, 0))
True

# Execute on specific date and time
>>> job = schedserver.Job('23 23 30 04 * hash', script_manager)
>>> job.has_to_run(datetime.datetime(2011, 4, 30, 23, 23, 12))
True
>>> job.has_to_run(datetime.datetime(1999, 12, 31, 23, 59, 59))
False
>>> job.has_to_run(datetime.datetime(2000, 1, 1, 0, 0, 0))
False
```

```
# Execute on last minute of the year
>>> job = schedserver.Job('59 23 31 12 * hash', script_manager)
>>> job.has_to_run(datetime.datetime(2011, 4, 30, 23, 23, 12))
False
>>> job.has_to_run(datetime.datetime(1999, 12, 31, 23, 59, 59))
True
>>> job.has_to_run(datetime.datetime(2000, 1, 1, 0, 0, 0))
False

# Execute on first minute of the year
>>> job = schedserver.Job('00 00 01 01 * hash', script_manager)
>>> job.has_to_run(datetime.datetime(2011, 4, 30, 23, 23, 12))
False
>>> job.has_to_run(datetime.datetime(1999, 12, 31, 23, 59, 59))
False
>>> job.has_to_run(datetime.datetime(2000, 1, 1, 0, 0, 0))
True

###################
### SchedServer ###
###################

>>> sched_server = schedserver.SchedServer(data_dir, script_manager)

################
### clean up ###
################

>>> shutil.rmtree(script_dir)
>>> shutil.rmtree(data_dir)
>>> shutil.rmtree(info_dir)

"""

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Packaging

Because the idea behind Viri, requires that it is installed in a large number of hosts, deployment needs to be as easy as possible. To make it possible, Viri is distributed using most common package formats for GNU/Linux distributions, RPM and DEB.

Also, because most common RedHat distributions does not include any package for version 3 of Python, packaging for Python itself has been implemented.

In this section there is the content of all files required to package Viri, and also Python.

RPM

.spec file for Viri:

```
%define name viri
%define version 0.1
%define release beta
%define python3_sitelib %(python3 -c "from distutils.sysconfig import get_python_lib;
print(get_python_lib())")
%define __prefix /usr

Name: %{name}
Version: %{version}
Release: %{release}
Summary: Remote execution of Python scripts (daemon)
Group: System Environment/Daemons
License: GPLv3+
URL: http://www.viriproject.com
Source: Viri-%{version}.tar.bz2
BuildArch: noarch
Requires: python3.1
Prefix: %{__prefix}
```

```
%description
Viri is an application to easily deploy Python scripts, tracking its
execution results. Viri has two different components, the virid daemon,
which should be installed on all hosts that will be managed, and the
viric command line utility. The client program viric can be used directly
by system administrators, but also can be integrated with third party
applications to automate tasks.
Some examples on what Viri can be useful for include data gathering,
synchronization of files, deployment of software; but it can be used
for everything which can be coded in the Python language.

%package client
Summary: Remote execution of Python scripts (client)
Group: Applications/System

%description client
Viri is an application to easily deploy Python scripts, tracking its
execution results. Viri has two different components, the virid daemon,
which should be installed on all hosts that will be managed, and the
viric command line utility. The client program viric can be used directly
by system administrators, but also can be integrated with third party
applications to automate tasks.
Some examples on what Viri can be useful for include data gathering,
synchronization of files, deployment of software; but it can be used
for everything which can be coded in the Python language.

%prep
%setup -n Viri-%{version}

%install
[ -d "$RPM_BUILD_ROOT" -a "$RPM_BUILD_ROOT" != "/" ] && rm -rf $RPM_BUILD_ROOT
make DESTDIR=$RPM_BUILD_ROOT os=redhat install

%post
read -p "Host code: " HOSTCODE
echo -e "\nHostCode: $HOSTCODE\n\n" >> /etc/viri/virid.conf
chkconfig virid --add
chkconfig virid on --level 2345
service virid start
```

```
%files
%defattr(-,root,root,-)
%doc AUTHORS LICENSE README
%{__prefix}/sbin/virid
%{python3_sitelib}/viri/__init__.py
%{python3_sitelib}/viri/rpcserver.py
%{python3_sitelib}/viri/schedserver.py
%{python3_sitelib}/viri/scriptmanager.py
/etc/viri/virid.conf
/etc/init.d/virid

%files client
%defattr(-,root,root,-)
%doc AUTHORS LICENSE README
%{__prefix}/bin/viric

%changelog
* Tue May 31 2011 Marc Garcia <garcia.marc@gmail.com> %{version}-%{release}
- Initial release
```

.spec file for Python 3.1

```
%define name python
%define version 3.1.3
%define binsuffix 3.1
%define libvers 3.1
%define release viri
%define __prefix /usr
%define libdirname %(( uname -m | egrep -q '_64$' && [ -d /usr/lib64 ] && echo lib64 ) ||
echo lib)

Summary: An interpreted, interactive, object-oriented programming language.
Name: %{name}%{binsuffix}
Version: %{version}
Release: %{release}
License: PSF
Group: Development/Languages
Source: Python-%{version}.tar.bz2
BuildRoot: %{_tmppath}/%{name}-%{version}-root
BuildPrereq: expat-devel
BuildPrereq: db4-devel
BuildPrereq: gdbm-devel
BuildPrereq: sqlite-devel
BuildPrereq: ncurses-devel
BuildPrereq: readline-devel
BuildPrereq: zlib-devel
BuildPrereq: openssl-devel
Prefix: %{__prefix}
Packager: Marc Garcia <garcia.marc@gmail.com>

%description
Python is an interpreted, interactive, object-oriented programming
language.  It incorporates modules, exceptions, dynamic typing, very high
level dynamic data types, and classes. Python combines remarkable power
with very clear syntax. It has interfaces to many system calls and
libraries, as well as to various window systems, and is extensible in C or
C++. It is also usable as an extension language for applications that need
a programmable interface.  Finally, Python is portable: it runs on many
brands of UNIX, on PCs under Windows, MS-DOS, and OS/2, and on the
Mac.
```

```
%changelog
* Sun Jun 12 2011 Marc Garcia <garcia.marc@gmail.com> [3.1.3-viri]
- Initial version, based on Python2.4 .spec

%prep
%setup -n Python-%{version}

%build
./configure --disable-ipv6 --with-pymalloc --prefix=%{__prefix}
make

%install
[ -d "$RPM_BUILD_ROOT" -a "$RPM_BUILD_ROOT" != "/" ] && rm -rf $RPM_BUILD_ROOT
mkdir -p $RPM_BUILD_ROOT%{__prefix}/%{libdirname}/python%{libvers}
make prefix=$RPM_BUILD_ROOT%{__prefix} install

# creating version binary symlinks
cd $RPM_BUILD_ROOT%{__prefix}/bin
rm -f python3
ln -s python%{binsuffix} python3
# Removing unwanted files
rm -f $RPM_BUILD_ROOT%{__prefix}/bin/python%{binsuffix}-config
rm -f $RPM_BUILD_ROOT%{__prefix}/bin/python3-config
rm -f $RPM_BUILD_ROOT%{__prefix}/bin/2to3
rm -f $RPM_BUILD_ROOT%{__prefix}/bin/idle3
rm -f $RPM_BUILD_ROOT%{__prefix}/bin/pydoc3
rm -rf $RPM_BUILD_ROOT%{__prefix}/include
rm -rf $RPM_BUILD_ROOT%{__prefix}/%{libdirname}/python%{libvers}/config
rm -rf $RPM_BUILD_ROOT%{__prefix}/%{libdirname}/python%{libvers}/idlelib
rm -rf $RPM_BUILD_ROOT%{__prefix}/%{libdirname}/python%{libvers}/lib2to3
rm -rf $RPM_BUILD_ROOT%{__prefix}/%{libdirname}/python%{libvers}/tkinter
rm -rf $RPM_BUILD_ROOT%{__prefix}/%{libdirname}/python%{libvers}/test
```

```
# Fixing header which refs to /usr/local/bin/python for
# compatibility with Solaris, but which breaks redhat
FIXFILE=$RPM_BUILD_ROOT%{__prefix}/%{libdirname}/python%{libvers}/cgi.py
TMPFILE=/tmp/fix-python-path.$$
echo '#!/bin/env python'"%{binsuffix}" > $TMPFILE
tail -n +2 $FIXFILE >> $TMPFILE
mv $TMPFILE $FIXFILE
$RPM_BUILD_ROOT%{__prefix}/bin/python%{binsuffix} \
$RPM_BUILD_ROOT%{__prefix}/%{libdirname}/python%{libvers}/py_compile.py \
$FIXFILE
$RPM_BUILD_ROOT%{__prefix}/bin/python%{binsuffix} -O \
$RPM_BUILD_ROOT%{__prefix}/%{libdirname}/python%{libvers}/py_compile.py \
$FIXFILE


%clean
[ -n "$RPM_BUILD_ROOT" -a "$RPM_BUILD_ROOT" != / ] && rm -rf $RPM_BUILD_ROOT


%files
%defattr(-,root,root)
%doc Misc/README Misc/cheatsheet Misc/Porting LICENSE Misc/ACKS Misc/HISTORY Misc/NEWS
%{__prefix}/share/man/man1/python%{binsuffix}.1.gz
%attr(755,root,root) %dir %{__prefix}/bin/python%{binsuffix}
%attr(755,root,root) %dir %{__prefix}/bin/python3
%{__prefix}/%{libdirname}/libpython%{libvers}.a
%{__prefix}/%{libdirname}/python%{libvers}
%{__prefix}/%{libdirname}/pkgconfig/python-3.1.pc
%{__prefix}/%{libdirname}/pkgconfig/python3.pc
```

DEB

changelog file:

```
viri (0.1) UNRELEASED; urgency=low

  * Initial release. (Closes: #XXXXXX)

 -- Marc Garcia <garcia.marc@gmail.com>  Tue, 31 May 2011 11:09:46 +0200
```

compat file:

```
8
```

control file:

```
Source: viri
Section: utils
Priority: optional
Maintainer: Marc Garcia <garcia.marc@gmail.com>
Build-Depends: debhelper (>= 8), python-support
Standards-Version: 3.9.1.0
Homepage: http://www.viriproject.com

Package: viri
Architecture: all
Depends: python3
Description: Remote execution of Python scripts
 Viri is an application to easily deploy Python scripts, tracking its
 execution results. Viri has two different components, the virid daemon,
 which should be installed on all hosts that will be managed, and the
 viric command line utility. The client program viric can be used directly
 by system administrators, but also can be integrated with third party
 applications to automate tasks.
 .
 Some examples on what Viri can be useful for include data gathering,
 synchronization of files, deployment of software; but it can be used
 for everything which can be coded in the Python language.
```

copyright file:

```
Viri was debianized by Marc Garcia on Tue, 31 May 2011 11:47:40 +0200.


Viri is available at http://www.viriproject.com


Copyright 2011 Marc Garcia <garcia.marc@gmail.com>


License:
    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 3, or (at your option)
    any later version.


    The text of the GPL can be found on Debian systems in
    /usr/share/common-licenses/GPL .
```

dirs file:

```
usr/bin
usr/share/pyshared/viri
etc/viri
etc/init.d
```

docs file:

```
AUTHORS
README
```

postinst file:

```
#!/bin/sh

set -e

# FIXME can't get debconf to work, so going the manual way for now
#. /usr/share/debconf/confmodule

case "$1" in
    configure)
        ln -s /usr/share/pyshared/viri /usr/lib/python3/dist-packages/

        #db_get viri/hostcode
        echo -n "Host code: "
        read RET
        echo "" >> /etc/viri/virid.conf
        echo "HostCode: $RET" >> /etc/viri/virid.conf
    ;;

    abort-upgrade|abort-remove|abort-deconfigure)
        rm -f /usr/lib/python3/dist-packages/viri
    ;;

    *)
        echo "postinst called with unknown argument \`$1'" >&2
        exit 1
    ;;
esac

#DEBHELPER#

exit 0
```

postrm file:

```
#!/bin/sh
set -e

case "$1" in
    purge|remove|upgrade|failed-upgrade|abort-install|abort-upgrade|disappear)
        rm -f /usr/lib/python3/dist-packages/viri
    ;;

    *)
        echo "postinst called with unknown argument \`$1'" >&2
        exit 1
    ;;
esac

#DEBHELPER#

exit 0
```

templates file:

```
Template: viri/hostcode
Type: string
Description: Host code
 Unique code of this host to be used by Viri scripts
```

rules file:

```
#!/usr/bin/make -f
# -*- makefile -*-
configure: configure-stamp
configure-stamp:
	dh_testdir
	touch configure-stamp
build: build-stamp
build-stamp: configure-stamp
	dh_testdir
	$(MAKE)
	touch $@
clean:
	dh_testdir
	dh_testroot
	rm -f build-stamp configure-stamp
	dh_clean
install: build
	dh_testdir
	dh_testroot
	dh_prep
	dh_installdirs
	$(MAKE) DESTDIR=$(CURDIR)/debian/viri install
binary-arch: build install
	# emptyness
binary-indep: build install
	dh_testdir
	dh_testroot
	dh_installchangelogs
	dh_compress
	dh_installdocs
	dh_installexamples
	dh_pysupport
	dh_link
	dh_fixperms
	dh_installdeb
	dh_shlibdeps
	dh_gencontrol
	dh_md5sums
	dh_builddeb
binary: binary-indep
.PHONY: build clean binary-indep binary install configure
```

Tables and Figures

Tables

Figures

References

1. Universitat Oberta de Catalunya (UOC, Open University of Catalonia), http://www.uoc.edu

2. Python programming language, http://www.python.org

3 UNIX operating system, http://www.unix.org / http://en.wikipedia.org/wiki/Unix

4. Remote login (rlogin) software utility, http://tools.ietf.org/html/rfc1282 / http://en.wikipedia.org/wiki/Rlogin

5. Remote shell (rsh) computer program, http://en.wikipedia.org/wiki/Remote_Shell

6. SSH network protocol, http://en.wikipedia.org/wiki/Secure_Shell

7. gexec remote execution system, http://www.theether.org/gexec/

8. Cfengine datacenter automation application, http://www.cfengine.org/

9. Nagios alert based monitoring system, http://www.nagios.org/

10. Ganglia statistical based monitoring system, http://ganglia.sourceforge.net/

11. Constructive Cost Model (COCOMO) algorithm for estimation of software projects, http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html

12. Comparison of verbosity between Java and Python http://codemonkeyism.com/comparing-java-and-python-is-java-10x-more-verbose-than-python-loc-a-modest-empiric-approach/

13. Ohloh open source public directory for software and people, http://www.ohloh.net

14. Free and Open Source Software (FOSS) software philosophy and licensing policies http://en.wikipedia.org/wiki/Free_and_open_source_software

15. Django web framework, http://www.djangoproject.com/

16. Remote Procedure Call (RPC), protocol category http://en.wikipedia.org/wiki/Remote_procedure_call

17. Common Object Request Broker Architecture (CORBA) messaging technology, http://www.corba.org/

18. Comparison of messaging technologies, http://www.ibm.com/developerworks/webservices/library/ws-pyth9/

19. XML-RPC specification, and Python implementation, http://www.xmlrpc.com / http://docs.python.org/release/3.1.3/library/xmlrpc.server.html

20. Extensible Markup Language (XML), http://www.w3.org/XML/

21. Hypertext Transfer Protocol (HTTP), http://www.ietf.org/rfc/rfc2616.txt / http://www.w3.org/Protocols/

22. Transport Layer Security (TLS) encryption protocol, http://datatracker.ietf.org/wg/tls/

23. Public Key Infrastructure (PKI), http://datatracker.ietf.org/wg/pkix/charter/

24. Secure Sockets Layer (SSL) protocol, http://en.wikipedia.org/wiki/Secure_Sockets_Layer

25. Python ssl module, http://docs.python.org/release/3.1.3/library/ssl.html

26. Object-Relational Mapping (ORM) software, http://en.wikipedia.org/wiki/Object-relational_mapping

27. CRUD (Create, Read, Update, Delete) operations, http://en.wikipedia.org/wiki/Create,_read,_update_and_delete

28. Git version control system, http://git-scm.com/

29. reStructuredText markup language, http://docutils.sourceforge.net/rst.html

30. http://sphinx.pocoo.org/